BUILDING OFFICE ADD-INS USING

OFFICE.JS

MICHAEL ZLATKOVSKY

Building Office Add-ins using Office.js

Michael Zlatkovsky

This book is for sale at http://leanpub.com/buildingofficeaddins

This version was published on 2017-08-27

Tweet This Book!

Please help Michael Zlatkovsky by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Building Office Add-ins using Office.js #buildingofficeaddins #addins #officejs #reading

The suggested hashtag for this book is #buildingofficeaddins.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#buildingofficeaddins

Contents

1. The book and its structure

1.1 The “evergreen”, in-progress book

This book is published using a “lean” methodology – publishing early, and publishing often.

Since writing the book is strictly my “moonlighting” activity – my day-job is to be a *developer* on Office.js APIs, not a technical writer – it will take a good long while before I am actually “done” with all the content I want to write about. I would much rather ship an **early** version of the book, addressing the most common questions and issues that I see folks struggling with, and then keep iterating on it from there.

In buying this e-book through LeanPub (where I’m also discounting it for the “early readers”), **you are entitled to free updates to the book**. So, expect to see more content appear month-to-month (and, if you sign up for email notification, to receive periodic emails updates about what new content got added).

I welcome any and all feedback about the writing, explanations, code-samples, and whatever else. I also welcome topic suggestions, if you see something I didn’t cover; or, if you see something that I *plan to cover*, but haven’t yet, just let me know and I can see if I can prioritize it. Section “*Bug reports / topic suggestions*” below will describe the best way for filing these sorts of issues/suggestions. Or, if you want to reach me directly, you can do so at michael@buildingofficeaddins.com.

Yours truly,

Michael Zlatkovsky, the author

1.2 Release notes

You are currently reading version **1.4** of the book. Release notes for major updates are posted on **http://buildingofficeaddins.com/release-notes**

Updates to the book are free for all existing readers (which is what lean-publishing, and the concept of an evergreen book, is all about!). Simply go to **https://leanpub.com/user_dashboard/library**, select the book, and download your newly-updated copy!

For any issues or topic requests, please file an issue on http://buildingofficeaddins.com/issues.

Version 1.6 (*August 26, 2017*) [261 pages]

- Major re-structuring of the book, splitting out the former chapters 5 & 6 – which were bursting at the seams – into a bunch of smaller chapters. Also streamlined the rest of the book, moving topics that were less immediately-necessary (e.g., API versioning) further towards the back of the book, to make it faster to get started.
- As part of the getting-started chapter, added a section for my recommendations of "*The optimal dev environment*"
- Expanded the section on "*Handling errors*".
- Added "*Recap: the four basic principles of Office.js*" to the chapter on core Office.js topics.
- Added runnable Script Lab snippets to "*Canonical code sample: reading data and performing actions on the document*" (including a refactored version that splits the task into multiple subroutines, and a plain ES5 JavaScript variant).

Version 1.5 (*August 2, 2017*) [249 pages]

- By popular demand, added a massively-detailed (15+ pages) and example-filled section on "Using objects outside the "linear" `Excel.run` or `Word.run` flow (e.g., in a button-click callback, in a `setInterval`, etc.). Its subsections include:

  - Re-hydrating an existing Request Context: the overall pattern, proper error-handling, `object.track`, and cleanup of tracked objects
  - A common, and infuriatingly silent, mistake: queueing up actions on the wrong request context
  - Resuming with multiple objects
  - Why can't we have a single global request context, and be one happy family?
- Addressed a couple of small reader-reported issues.

Version 1.4 (*July 19, 2017*) [232 pages]

- Added a detailed section for how to check whether an object exists (e.g., whether there's a worksheet by a particular name, whether two ranges intersect, etc). The section also talks about the powerful but unusual "null object" pattern, used in methods and properties suffixed with `*OrNullObject`.
- Added a section with links to API documentation resources.
- Addressed a couple of small reader-reported issues.

Version 1.3 (*May 23, 2017*) [217 pages]

- Added an entire section devoted to the `PropertyNotLoaded` error.
- Added info about Script Lab, a playground tool to easily try out code samples described in this book
- Addressed a variety of small reader-reported issues.

Version 1.2 (*Feb 20, 2017*) [210 pages]

- Added a topic on the different flavors of Office 2016 / Office 365 – and the practical implications for developers.
- Added a topic on API Versioning and Requirement Sets.
- Greatly expanded the "TypeScript-based Add-ins" topic, adding instructions for the updated Yeoman generator.
- Added a topic for attaching the debugger to Add-ins (breakpoints, DOM explorer, etc.)
- Added a link to the book's companion Twitter account.
- Addressed a number of other reader-reported issues.

Version 1.1 (*Jan 22, 2017*) [189 pages]

- Re-pivoted the book around TypeScript and the `async/await` syntax. Moved the JS-specific content to a separate Appendix
- Added an information-packed JavaScript & TypeScript crash-course, tailored specifically at Office.js concepts, for those who are new to the world of JS/TS.
- With TypeScript now a first-class citizen of the book, added "*Getting started with building TypeScript-based Add-ins*" (section 3.2). I expect to continue to expand this section in future releases.
- Added an in-depth explanation of the internal workings of the Office.js pipeline and proxy-object model. See "*Implementation details, for those who want to know how it really works*" (section 5.5).
- Re-arranged, edited, and added to the content of the "*Office.js APIs: Core concepts*" chapter (chapter 5).
- Added links to downloadable code samples, for a few of the larger samples.
- Addressed a variety of smaller feedback items.

Version 1.0 (*Dec 12, 2017*) [144 pages]

Initial release.

1.3 Bug reports / topic suggestions

Having now had experience in both, I think that writing an [evergreen] book is akin to writing an [evergreen] add-in / website. Try as you may, there will be bugs; and there will also be not-yet-implemented features, or new ideas that simply hadn't occurred before.

To this end, I'd like to provide readers with a way to easily log and track content issues and topic suggestions. Issues can be:

- Simple content issues: A misspelling, an incomplete phrase, a sentence that no longer makes sense.
- Requests for additional detail in *existing* topics.
- Requests for *brand new topics*. I might already be planning to write something about it eventually, but this will let you subscribe to finding out when the content is made available; and will also help me gauge topic interest.
- Issues with sample code or related assets: code is unclear; the design is sub optimal; something needs a comment; or perhaps an existing comment needs to be clarified or removed.
- Anything else?

The issue repo can be accessed through **http://buildingofficeaddins.com/issues**

1.4 Twitter

As reader of the evergreen book, you will receive periodic updates from LeanPub when I publish a major version (unless you opt out, that is). I expect to send out such communications once every month or two.

If you'd like to receive more frequent status updates – both about the book, articles that I put up on my site, interesting StackOverflow questions, or re-tweets of interesting updates about Office Add-ins that I find on the web – I encourage you to follow my twitter feed at

https://twitter.com/BuildingAddins

or view the feed embedded on my site, at

http://buildingofficeaddins.com/tweets/

By the same token, if you blog or tweet about the book, I would be much obliged if you can use the tag **#buildingofficeaddins** and/or **officejs**, and also @mention me: **@BuildingAddins**.

1.5 Who should read this book

This book is aimed at the *professional developer* who is tasked with creating an Office Add-in (a.k.a an *Office Web Add-in*, and formerly known as an *App for Office*). In particular, it is aimed at the "*new*" Office 2016+ wave of Office.js APIs – which, at the time of writing, is supported in Word, Excel, and OneNote[1].

Office has a rich and wonderful legacy of programmability, with VBA spanning the full gamut of developer skill levels – from novice programmers tweaking simple macros, to professional developers writing complex Office customization. But for purposes of this book (and due to the more complex nature of the underlying Office Add-ins platform – for now, anyway), it is really the *professional developer* that is the target audience. With that definition, I mean someone who is well-versed in code, who is eager to learn something new, and who is unfazed by occasional difficulty (which you will naturally get, being on the cutting edge of a developing Office Add-ins platform). A working knowledge of JavaScript is a plus, as is power-user knowledge of the Office application that you are targeting.

Importantly, this book is *not* meant as a "standard" API reference manual, which might walk you through a series of tasks or APIs, focusing on the particulars. From my perspective, we have dozens of object types, and hundreds of methods and properties, in each of the Office hosts, all of these are dutifully documented in our online API reference documentation. To put these in print, in the form of a static book, would serve very little use.

Instead, this book is about the underlying principles that these APIs, however diverse, have in common. It is about the conceptual threads that bind the API surface area with the common runtime that they share. My goal is to paint the overarching picture of the API model, and to zoom in on details that transcend beyond any given API object. In this regard, this book is about going

[1] As of December 2016, Word, Excel, and OneNote have all adopted the *new 2016 wave* of Office.js APIs. Outlook – though steadily continuing to expand its APIs – is continuing to use the "Office 2013" style of APIs even in its Office 2016 applications (partially because many of its scenarios are more about data consumption, and less about object-by-object automation, which is where the new model would shine). PowerPoint and Access have, so far, remained unchanged in their API surface relative to 2013; and Project, though it has added a number of APIs, is also still using the "Office 2013" style.

from copy-pasting online sample code, to understanding the key concepts that make these samples work. In short, this book is about writing effective, efficient, testable code, and the tools that will help you do so.

As you’ll read in the introduction to Office Add-ins, Office 2016 – and, of course, its Office Online & Mac & iOS equivalents – represents a major re-birth of the APIs, with each of the supported hosts getting a custom-tailored object model that dives far deeper than the original set of Office 2013’s “common APIs”. This *new* (2016+) wave of Office.js APIs is the subject of this book. You may find that a part of the content in this book â€“ especially the more general topics like an introduction to Office Add-ins, prerequisites, debugging techniques, pro tips, and various “beyond-API” topics â€“ will apply to both the 2013 and 2016 Office API models, but the book’s focus and *raison d’etre* is the *new Office 2016 wave* of Office.js APIs.

1.6 From the author

This book is written for Office Developers, by an Office Developer. By the latter, I mean both that I had been a developer who used Office technology extensively for a number of years, *and* that I am now a developer on the Office team at Microsoft.

While generationally I am probably closer to the Web-happy, Angular-2-happy, React-happy crowd, I began my journey into Office programming through good old VBA. I have great fondness for VBA: not just because Visual Basic 6 was my first language (which a professor of mine once compared to “first love”), but also because of how *easy* it was to get started with it. I used VBA for a project that became the basis of a small company that my uncle and I co-founded, creating a market-research package that was based purely on Excel. A couple of years later, as our solution continued to grow, I began the non-trivial task of porting portions of the code to VSTO (Visual Studio Tools for Office) and the .NET Framework. The task, done piecemeal over the course of two years, was effectively a complete re-write of the add-in: so I can appreciate first-hand the work involved in switching to a new language and methodology.

For VBA and VSTO, there are a great many reference materials, both in the forms of books, blogs, MSDN forum questions, and StackOverflow questions. For example, in transitioning from VBA to VSTO, I was greatly helped in that transition by a couple of excellent books, “VSTO for Mere Mortals”[2] and “Visual Studio Tools for Office 2007”[3]. By contrast, for Office Add-ins, the amount of available information is quite minimal: effectively just MSDN documentation for particular APIs, and questions on StackOverflow where I am an active participant[4]. Having been part of the design and implementation

[2] See “VSTO for Mere Mortals: A VBA Developer’s Guide to Microsoft Office Development Using Visual Studio 2005 Tools for Office” by Kathleen McGrath and Paul Stubbs.

[3] See “Visual Studio Tools for Office 2007: VSTO for Excel, Word, and Outlook” by Eric Carter and Eric Lippert.

[4] See questions tagged as “**[office-js]**” on StackOverflow – which you can get view via **http://stackoverflow.com/questions/tagged/office-js.** For questions **answered by me**, search StackOverflow using the keywords “**user:678505 [office-js]**”. At the time of writing (October 2016), I have answered 118 questions out of the 495 that have been asked to date.

of the *new* Office.js APIs from the very start, I hope to share some insights & techniques – and above all, the full end-to-end story – of how to create Add-ins using the new Office.js.

1.7 A few brief notes

A few brief notes before we get started:

- This book assumes some general programming experience, and ideally a working knowledge of JavaScript. "Chapter 4. JavaScript & Promises primer (as pertaining to our APIs)" can get you somewhat up to speed, but if you're intending to develop add-ins professionally, you'll likely want a more formal JS training / book.
- While you may need various other web skills (HTML & CSS expertise, experience writing back-end services and databases, and so forth) in order to create the full-fledged Add-in, these topics are not Office Add-in specific, and are covered in countless books and web articles. Beyond the occasional reference or link – or just enough explanation to show an example – this book will focus mostly on the "*Office.js*" part of writing Office Add-ins.
- You should have a reasonably working knowledge (and ideally, "Power User" knowledge) of the host application that you will be targeting. Here and elsewhere throughout the book, "host application" refers to the Office application that your add-in will be running in (Word, Excel, etc.). After all, the host application is what your end-users will use, and so you'll need to have the expertise to test that it all works under all circumstances! An example: in Excel, worksheet names can only contain up to 31 characters, and may not include a number of reserved characters. This means that, to avoid having the application stop midway through an operation, you will want to pre-validate the user input, or handle the error through other means.
- The concepts and tools throughout this book are applicable to all of the Office applications that have adopted the "new" Office.js model (which, at the time of writing, is Excel, Word, and OneNote). In this book, I will generally use Excel-based samples, both because I love Excel, and because – at least for VSTO – more developers were programming against Excel than against any of the other hosts. But again, if you substitute *Word* or *OneNote* in place of *Excel* in the sample code – and if you substitute in the analogous objects – the exact same concepts will apply. I'll try to sprinkle in some Word examples too, for fairness' sake.

- Finally, while the book is written by someone who works on the Office Extensibility Platform team at Microsoft, the information and viewpoints presented in this book represent the *viewpoints of the author*, not of Microsoft as a company.

1.8 Acknowledgments

First and foremost: the new Office.js paradigm would have been impossible without the vision and incredible technical skills of **Shaofeng Zhu**, Principal Software Engineering Manager on the Office Extensibility Platform team. Shaofeng paved the way in prototyping the design, in creating a codegen pipeline that automatically generates the JavaScript and much of the internal glue, and then in leading the Excel programmability team to ship our first ExcelApi 1.1 – while simultaneously assisting Word, OneNote, and others to jump onto the new paradigm. Shaofeng is the unsung hero of the Office.js APIs for Office 2016 and beyond – so I'd like to take this opportunity to give credit where credit is due.

I have had a number of people provide feedback in successive edits of this book. These include:

- **Ricky Kirkham**, Content Developer on the Office Extensibility Platform team, and an early reader/reviewer/enthusiast for the book, who provided insightful feedback for every chapter, every section, and quite likely every page therein.
- **Alexey Butenko**, former colleague who spent several years deep in the bowels of the Excel and Word object-models… and came up with a number of innovative approaches for performance-optimizing VSTO code along the way. He's too contributed an incredible amount of insightful feedback on the book.
- **Jakob Nielsen**, Principal Designer on the Excel team – who unknowingly instigated my book-writing effort by casually tossing out *"you know, someone should take all these answers that Michael's putting on StackOverflow, and make a book out of it…"* Thanks for the nudge, Jakob!
- **Shahar Prish**, Principal Software Engineer on Power BI (and formerly a developer on the Excel team), with a keen eye for technical inaccuracies.
- **Mike Ammerlaan**, Director of the Product Marketing Ecosystem for Office Add-ins, who provided some stylistic feedback on the book.
- **Siew Moi Khor**, Content Developer on the Office Extensibility Platform team, who has long been involved in Office Programmability, and who also helped line up some of the above reviewers.

I would also like to thank **Seyit Yilmaz**, a Designer on Excel team, who created the book’s cover image.

Many thanks to all of you!

2. Introduction to Office Add-ins

Chapter structure & planned content

Note: Sections that have already been written, and are **included** in this book, are **bolded and hyperlinked**; the rest are planned topics, but have not been written (or polished) yet.

- Office Add-ins (a.k.a. Office Web Add-ins)
  - What are they, and how are they different from other Office Extensibility technologies?
  - Web Add-ins? What is meant by “web”, and what technologies are involved?
  - Types of Office Web Add-ins (content add-ins, task-pane add-ins, add-in commands)
- **What’s new in the Office 2016 APIs (relative to 2013)?**
- **What about VBA, VSTO, & COM add-ins?**
- **“But can Office.js do XYZ?”**
- **A word on JavaScript and TypeScript**
- **Office.js: The asynchronous / deferred-execution programming model**
  - **Why is Office.js async?**
  - **What is meant by “the server”?**

2.1 What's "new" in the Office 2016 APIs (relative to 2013)?

As you'll read in greater detail in *"Office.js: The asynchronous / deferred-execution programming model*", the Office.js API model was first introduced in Office 2013, under the name of "Apps for Office". The model was an ambitious – if somewhat limited –- attempt to create a new kind of web technology based APIs, and have those APIs work cross platform and across multiple host applications.

Office 2016 has seen a complete overhaul of the original 2013 API model, creating new and host specific APIs. These APIs offer a rich client-side object model that accurately reflects the unique characteristics of each object type for each of the hosts. For example, the Excel APIs have classes for Worksheets, Ranges, Tables, Charts, and so forth, with each object type offering dozens of properties and methods. Similarly, the Word API now have the notion of Paragraphs, and Content Controls, and Images, and more. The team has also changed the overarching design process, building out the APIs with great transparency in the form of "open specs", which are posted online for the community to give feedback on before implementation ever takes place. In a similar vein, several colleagues on my team – and an increasing number of folks from the external community – are active participants on StackOverflow, where we actively monitor any questions that are tagged with "**office-js**".

The Office 2016 release also signifies a much deeper commitment to cross-platform consistency in the APIs. For example, all of the new APIs in Excel that are available in Office 2016 for Windows are also available in Excel Online, Excel for Mac and Excel on iOS. To keep up with developers' demands, the new APIs are shipped continuously, usually once per quarter, with those APIs becoming immediately available to users of Office 365[1] on the Desktop, on Office Online, and Mac/iOS.

Even beyond the APIs, Office 2016 offers enhancements to other parts of the overall programming framework – most notably, the ability for Office Add-ins to appear in the ribbon and to launch dialog windows. If you were previously

[1] For more information on the "MSI" vs. subscription-based Office installations, see *"Office versions: Office 2016 vs. Office 365 (Click-to-Run vs. MSI), Deferred vs. Current channel"*.

on the fence for whether to look into Office Add-ins, I hope that the APIs described in this book will help change your mind.

2.2 What about VBA, VSTO, and COM Add-ins?

Microsoft Office has a great legacy of programmability. For many, this is primarily known through VBA – Visual Basic for Applications. For others, it is known through the VSTO (Visual Studio Tools for Office) or through writing COM Add-ins.

To be very clear: none of these technologies are going away anytime soon. That being said, VBA is only supported on Windows desktop and Mac (with VSTO & COM only on Windows desktop), making it impossible for any of these customizations to run on Office Online or on the iPad. Moreover, in the case of VBA, the underlying technology is almost two decades old, making it difficult to create beautiful UI, to connect to online services, or to utilize the web oriented knowledge and skills of emerging developers.

Are Office Add-ins the answer to everything? No, or at least not yet. VBA and VSTO still offer many APIs and functionalities that are not available in Office.js. The Office Extensibility Team – of which I am a member – is actively working on closing the gap, but there's no denying that there's still a long ways to go. And, for now, VBA still remains the single most accessible tool for nonprofessional programmers to easily create a simple script to automate their everyday Office tasks.

If you have an existing solution that uses VBA for VSTO, and you are happy with your customer base and desktop only platform support, there is little reason for you to switch platforms, for now. Office.js will require some re-thinking of your add-in, and will require learning new concepts in place of the ones that are already familiar to you. The investment, for now, may or may not be worth it. On the other hand, if you're creating an Office extension for the very first time, I would strongly encourage you to give Office Add-ins a try.

As a company, Microsoft has committed to Office Add-ins as the next wave of Office Programmability, so Office Add-ins are definitely here to stay. If you want to be part of the emerging present and the upcoming future in Office Programmability, this book is for you.

For a more detailed comparison of Office Add-ins, VSTO, and VBA, see https://blogs.msdn.microsoft.com/officeapps/2013/06/18/roadmap-for-apps-for-office-vsto-and-vba/. The blog is from a few years back, and was written by yours truly – but it the comparisons it offers are still relevant today (and note that the new Office 2016 wave of APIs and the Ribbon & Dialog UX functionality only increases the scope of what Office Add-ins can do, even if they couldn't in the Office 2013 incarnation).

2.3 “But can Office.js do XYZ?”

I frequently hear the question: “In VBA, I could do XYZ. Can I do it in Office.js?” The answer, however unsatisfactory it may be, is the usual “it depends”.

From the standpoint of API richness, Office.js is undoubtedly constrained relative to VBA, simply because it’s an evolving platform – and evolution takes time. The platform is not there yet with all of the APIs. In fact, it might *never* getthere – some APIs might just be deemed too dangerous, or too platform-specific, to expose. Instead, the Extensibility team has tried to enable key scenarios, and to provide “families” of related API functionality (i.e., you’ll see a lot of depth to the Excel Range object, and to Worksheets, and Tables – but then nothing for Comments or Shapes yet, because it was deemed more important to provide deep high-quality APIs for a smaller subset of objects, rather than spreading thin by trying to shallowly deliver on too much).

On the UX front, it’s more of a mix. With Office Add-ins, it’s true that the developer doesn’t have as much control over the UX (dialogs and ribbon extensibility have been added in Office 2016, but still not to the same rich extent as VSTO). On the other hand, I would claim that the UX you can achieve *within* a given taskpane is an order of magnitude better than in VBA, and a fair bit better than a VSTO solution: simply because you can use all of the rich UX of web design, and easily incorporate libraries and frameworks that offer fabulous UX. I think this is a real strength of Office Add-ins (as is the appeal of re-using your HTML/CSS web assets and tools, and developmental skills)

Where Office Add-ins shine is the **cross-platform story**, the **ease of acquisition** from the Office Store, and the **ability to deliver instant updates** to add-in code. If you need any of these three, and particularly the cross-platform bit, Office Add-ins are likely the way to go, even if you might not have all the API functionality that you wish you had. Also, remember than an API that is missing today might well be on the roadmap for tomorrow – both me personally, and the team at large, welcome your feedback with helping prioritize those (see section “*Contributing to the API design process*”!) On the other hand, if you care nothing about cross-platform or the Office Store, and are deploying a desktop-only solution on IT-issued computer where you can control anything through Group Policy, VBA or VSTO might well be better choices for now, as they generally would offer a more familiar programming model and richer APIs.

In short, do what’s right for your situation and your customers – but do give Office Add-ins some serious thought, especially if portability, ease of installs and updates, and the ability to re-use existing web content is important to you!

2.4 A word on JavaScript and TypeScript

Office Add-ins are built using web technologies. This means that, at least for interacting with the document, you'll be using JavaScript – lots of it.

JavaScript is a very curious language. It is obviously immensely popular: in StackOverflow's developer survey for 2016[2], it is far and away the most popular language for full-stack and front-end developers, and (though by a smaller margin), is the most popular language even for *back-end* developers. And yet, relative to the "standard" languages like C# or Java, the language is incredibly quirky! No type safety, no out of the box classes, two types of equality comparisons (== vs. ===), and the list goes on. How do you program in such an environment?

Coming from the strong and comfortable type safety of C#, VB.NET, and Java – complete with .NET's LINQ (Language-Integrated Query Language) and other runtime goodies – I was frustrated when first encountering JavaScript. Actually, "frustrated" doesn't even begin to describe it. I distinctly remember biking in the rain from work one winter day. I was cold and wet, but one quizzical thought still managed to surface to my consciousness: "I wonder what's more miserable: biking through this sort of weather, or programming in JavaScript?"

Fortunately, bit by bit, JavaScript grew on me. I actually find it quite delightful to program in it now, and appreciate the flexibility that if affords. But I still discover interesting new nuances and surprises (not always pleasant) about it every day. There is a reason for a book series called "You Don't Know JavaScript"[3], aimed at *the professional developer*!

Still, one thing I could never get used to in JavaScript was the complete lack of type safety (and reliable type inferencing, for IntelliSense's sake). Why couldn't JavaScript alert me of obvious mistakes, such as using incompatible types, or misspelling a property name? And why – despite Visual Studio's best efforts with JavaScript pseudo-execution at design time – could I not easily[4]

[2] http://stackoverflow.com/research/developer-survey-2016.

[3] See https://github.com/getify/You-Dont-Know-JS/blob/master/up%20&%20going/ch3.md for a good summary of what you may or may not know.

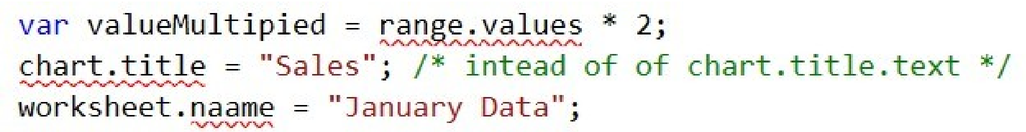
[4] Technically-speaking, there are a couple of little-known ways to get the JavaScript engine to cooperate with you in passing parameter types into functions, but there is definite effort required. See section "*JavaScript IntelliSense*".

get the type information to flow into my functions?

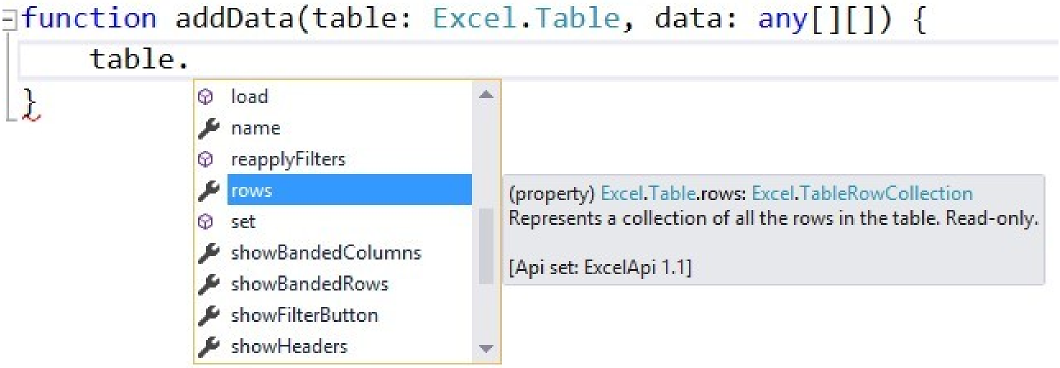
Enter TypeScript, a language developed by Microsoft's leading language experts, in an open-source, agile, "new Microsoft" fashion. TypeScript made its public debut in October 2012, and has been gaining popularity ever since. TypeScript is a typed superset of JavaScript that compiles into plain and idiomatic JavaScript. As http://www.typescriptlang.org explains:

> ***Starts and ends with JavaScript***
>
> *TypeScript starts from the same syntax and semantics that millions of JavaScript developers know today. Use existing JavaScript code, incorporate popular JavaScript libraries, and call TypeScript code from JavaScript.*
>
> *TypeScript compiles to clean, simple JavaScript code which runs on any browser, in Node.js, or in any JavaScript engine that supports ECMAScript 3 (or newer).*
>
> ***Strong tools for large apps***
>
> *Types enable JavaScript developers to use highly-productive development tools and practices like static checking and code refactoring when developing JavaScript applications.*
>
> *Types are optional, and type inference allows a few type annotations to make a big difference to the static verification of your code. Types let you define interfaces between software components and gain insights into the behavior of existing JavaScript libraries.*
>
> ***State of the art JavaScript***
>
> *TypeScript offers support for the latest and evolving JavaScript features, including those from ECMAScript 2015 and future proposals, like async functions and decorators, to help build robust components.*
>
> *These features are available at development time for high-confidence app development, but are compiled into simple JavaScript that targets ECMAScript 3 (or newer) environments.*

Personally, I love TypeScript. And in the case of Office.js, I think it is quite indispensable for writing non-trivial code. Take a look at a screenshot of some simple code snippets in Visual Studio, with all four of the pain-points I mentioned earlier. With TypeScript, each of the first three statements are highlighted as errors, and by specifying my parameter types (e.g., "`table: Excel.Table`"), I can keep the full power of IntelliSense even when passing objects across function boundaries.



```
var valueMultipied = range.values * 2;
chart.title = "Sales"; /* intead of of chart.title.text */
worksheet.naame = "January Data";
```



```
function addData(table: Excel.Table, data: any[][]) {
    table.
}
```

(property) Excel.Table.rows: Excel.TableRowCollection
Represents a collection of all the rows in the table. Read-only.

[Api set: ExcelApi 1.1]

IntelliSense across functions, by telling TypeScript what type I expect

But the thing that won me over completely to TypeScript – not for myself, as I was already a fan, but rather for readers of this book – is TypeScript 2.1, and the introduction of the `async/await` keyword. When I started writing the book in the Fall of 2016, `async/await` had not yet shipped[5]. I would still recommend TypeScript – for IntelliSense, compile-time safety, template strings, and more – but I had kept the book firmly rooted in JavaScript, feeling that it is the *lingua franka* of the web, and that someone's choice to opt into TypeScript should be a purely personal preference. The introduction of `async/await` has turned the table, though, to the point that I now firmly believe that new Office 2016 APIs

[5] More accurately, the *downlevel compilation* of async/await to EcmaScript 5 hadn't yet shipped. Async/await could already be used with ES6 (i.e., for Node programming), but without compiling down to EcmaScript 5, it was of little use to websites and add-ins. (For the record, Add-ins on the Desktop presently use Internet Explorer 11 (or lower, if the computer doesn't have IE 11) – so whatever programming you do, it needs to be targeting EcmaScript 5).

and TypeScript were a match made in heaven – and that *not* using `async/await` and the rest of the TypeScript features is lunacy. Once you've tried it, I don't think you'll ever look back!

To this end, I will use TypeScript notation liberally through this book, as it makes the model easier to explain and comprehend. TypeScript is a superset of JavaScript, so there are only a handful new concepts to learn relative to JS (and a bunch of JS-specific ones that you'll no longer need to worry about!). I will do a very quick crash-course on TypeScript concepts in the Prerequisites chapter, which should set you up with everything you need to know for purposes of this book. For those who are determined to avoid the TypeScript/ES6/async-await bandwagon, please see *"Appendix A: Using plain JavaScript instead of TypeScript*", which covers JS-specific techniques and also shows JS "translations" of some of the key code samples.

> Note: Most of the concepts in TypeScript are part of the *future* of JavaScript: usually the EcmaScript 6 specification, occasionally ES7 specification. In practice, though, because Add-ins run on Internet Explorer when on the Desktop (usually IE 11) – and because even IE 11 doesn't support EcmaScript 6 concepts – your common denominator will be ES5, which lacks many of the excellent features that are coming to JavaScript *eventually*. Hence, when I say that "TypeScript offers support for *xyz*", that's not to say that ES6 doesn't – but just that if you want to use it with Add-ins, letting TypeScript compile the feature down to ES5 is far and away the easiest.

For a no-setup (may I say delightful?) way to try out TypeScript with Office.js – including the beauty of using `async/await` – you can try it using Script Lab (see section "*Script Lab: an indispensable tool*"). Script Lab, a coding & learning playground for Office.js, accepts both JavaScript and TypeScript input, and you'll find that most of its samples make good use of the TypeScript features listed above.

Once you've tried out TypeScript in this setup-less Script Lab environment, section "*Getting started with building TypeScript-based add-ins** offers step-by-step guidance for getting Office.js and TypeScript to play together in a real project. It only takes a couple minutes to set up – so if you are developing a large and complex application, and/or if you find that the

JavaScript IntelliSense engine in Visual Studio isn't giving you the type of support that you'd like, I highly encourage you to give TypeScript a try.

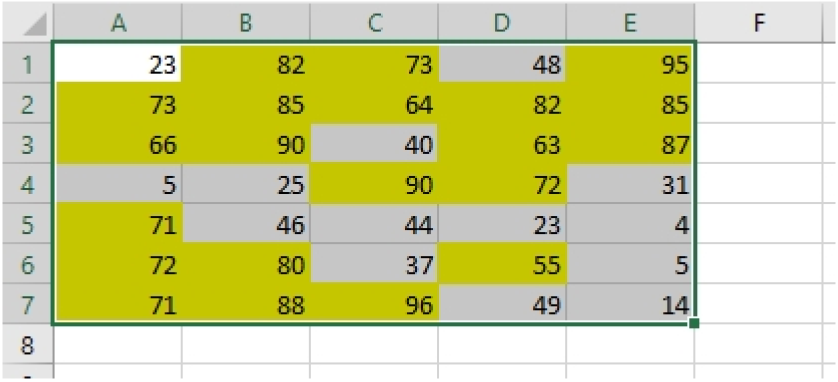
2.5 Office.js: The asynchronous / deferred-execution programming model

2.5.1 Why is Office.js async?

For those who have used VBA before, you will know that VBA code was always executed in a linear (synchronous) fashion. This is very natural for an automation task, where you're essentially manipulating a document through a series of steps (and where, more often than not, the steps are similar to the sequential series of steps that a human would do). For example, if you needed to analyze the current selection in Excel and highlight any values that were greater than 50, you might write something like this:

VBA macro for highlighting values over 50

```
Dim selectionRange As Range
Set selectionRange = Selection
Call selectionRange.ClearFormats

Dim row As Integer
Dim column As Integer
Dim cell As Range

For row = 1 To selectionRange.Rows.Count
    For column = 1 To selectionRange.Columns.Count
        Set cell = selectionRange.Cells(row, column)
        If cell.Value > 50 Then
            cell.Interior.Color = RGB(255, 255, 0)
        End If
    Next column
Next row
```



	A	B	C	D	E	F
1	23	82	73	48	95	
2	73	85	64	82	85	
3	66	90	40	63	87	
4	5	25	90	72	31	
5	71	46	44	23	4	
6	72	80	37	55	5	
7	71	88	96	49	14	
8						

Screenshot of the code in action

When run, such macro would execute line by line, reading cell values and manipulating them as it went. The macro have complete access to the in-memory representation of the workbook, and would run almost at the native Excel level, blocking out any other native Excel operations (or, for that matter, any user operations, since the VBA code executes on the UI thread).

With the rise of .NET, VSTO – Visual Studio Tools for Office – was introduced. VSTO still used the same underlying APIs that VBA accessed, and it still ran those APIs in a synchronous line-by-line fashion. However, in order to isolate the VSTO add-in from Excel – so that a faulty add-in would not crash Excel, and so that add-ins could be resilient against each other in a multi-add-in environment – VSTO had each code solution run within its own fully-isolated AppDomain. So while .NET code itself ran very fast – faster than VBA by pure numbers – the Object-Model calls into Excel suddenly incurred a significant cost of cross-domain marshaling (resulting in a ~3x slowdown compared to VBA, in my experience).

Thus, the VBA code above – translated into its VB.NET or C# equivalent – would continue to work, but it would run far less efficiently, since each subsequent *read* and *write* call would have to traverse the process boundary. To ameliorate that, the VSTO incarnation of the code could be made significantly faster if all of the *read* operations were lumped into a single *read* call at the very beginning. (The *write* operations, of setting the background of each cell one-by-one, would still have to remain as individually-dispatched calls[6]).

Here is C# Version of the code that addresses the above scenario, but this time reads all of the cell values in bulk (see line **#4** below).

VSTO incarnation of the code, with bulk-reading of the selection values (line #4)

```
 1  Range selectionRange = Globals.ThisAddIn.Application.Selection;
 2  selectionRange.ClearFormats();
 3
 4  object[,] selectionValues = selectionRange.Value;
 5
 6  int rowCount = selectionValues.GetLength(0);
 7  int columnCount = selectionValues.GetLength(1);
 8
 9  for (int row = 1; row <= rowCount; row++)
10  {
11      for (int column = 1; column <= columnCount; column++)
12      {
13          if (Convert.ToInt32(selectionValues[row, column]) > 50)
14          {
15              Range cell = selectionRange.Cells[row, column];
16              cell.Interior.Color = System.Drawing.Color.Yellow;
17          }
18      }
19  }
```

Note that the VSTO code, when interacting with the documents, still runs on (a.k.a., “blocks”) the Excel UI thread, since – due to the way that Excel (and Word, and others) are architected – all operations that manipulate the document run on the UI thread. But fortunately for VSTO, the process-boundary cost – while an order of magnitude higher than that of VBA – is still relatively small in the grand scheme of things, and the batch-reading

[6] Technically not 100% true, since you could create a multi-area range and perform a single “write” operation to it – but there is a limitation on how many cells you can group together, and the performance boost is still not nearly as good as what the VBA performance would have been. There are some APIs, like reading and writing to values or formulas, that can accept bulk input/output, but most of the rest – like formatting – must be done on a range-by-range basis.

technique described above can alleviate a chunk of the performance issues… and so, VSTO could continue to use the same APIs in the same synchronous fashion as VBA, while letting developers use the new goodness of the .NET Framework.

With the introduction of Office Add-ins (known at the time as "Apps for Office") in Office 2013, the interaction between the add-in and the host application had to be re-thought. Office Add-ins, based in HTML and JavaScript and CSS, needed to run inside of a browser container. On Desktop, the browser container is an embedded Internet Explorer process[7], and perhaps the synchronous model could still have worked there, even if it took another performance hit. But the real death knell to synchronous Office.js programming came from the need to support the Office Online applications, such as Word Online, Excel Online, etc. In those applications, the Office Add-ins runs inside of an HTML iframe, which in turn is hosted by the *parent* HTML page that represents the document editor. While the iframe and its parent are at least part of the same browser window, the problem is that the bulk of the documents might not – and generally is *not* – loaded in the browser's memory [8]. This means that for most operations [9], the request would have to travel from the iframe to the parent HTML page, all the way to an Office 365 web server running in a remote data center. The request would then get executed on the server, which would dutifully send the response back to the waiting HTML page, which would pass it on to the iframe, which would finally invoke the Add-in code. Not surprisingly, such round-trip cost is not cheap.

[7] If you're curious for how the interaction between Office and the embedded Internet Explorer control is done: it is through a `window.external` API that IE provides, which acts as the pipe between IE and the process that created it. The same technique is possible in .NET and WinForms/WPF applications as well. See https://msdn.microsoft.com/en-us/library/system.windows.forms.webbrowser.objectforscripting(v=vs.110).aspx for more info on the latter.

[8] Imagine a 50MB Word document, complete a bunch of images. Does it make sense for the browser to receive all 50MB at once, or could it progressively load only the pages that it needs in the neighboring vicinity, and only serve up the compressed and optimized, rather than raw, copies of the images?

[9] The actual amount of code that can be run locally (i.e., does not require a roundtrip to the server) varies greatly depending on the host application. On one extreme end of the spectrum, Excel Online requires that pretty much *all* operations are executed remotely. On the opposite side, OneNote Online has a pretty good local cache of the document, and needs to call out to the server much less frequently.

To put it into perspective: imagine that the entire roundtrip described above takes 50 milliseconds, and we are running a hypothetical synchronous and JavaScript-icized version of the VSTO macro. Imagine that we have one hundred cells, of which 50 meet the criteria for needing to be highlighted. This would mean that we need to make one request to clear the formatting from the selection, another to fetch all of the data, and then 50 requests for each time that we set on individual cell's color. This means that the operation would take (2 + 50) * 50 milliseconds, or just over 2.5 seconds. Perhaps that doesn't sound all that terrible… but then again, we were operating on a mere 100 cells! For 1000 cells, we'd be looking at 25 seconds, and for 10,000 cells we would be at over four minutes. What would the user be doing – other than sitting back in this chair and sipping coffee – while waiting for the Add-in operation to complete?!

If synchronous programming was out, the only remaining choice was asynchrony. In the **Office 2013** model, this was embodied by a whole bunch of methods that ended with the word "Async", such as:

```
    Office.context.document.setSelectedDataAsync(data, callback);
```

In this **Office 2013** design, ***every operation was a standalone call*** that was dispatched to the Office host application. The browser would then wait to be notified that the operation completed (sometimes merely waiting for notification, other times waiting for data to be returned back), before calling the callback function.

Thus, while the Office 2013 APIs solved the Async problem, the solution was very much a request-based Async solution, akin to server web requests, but not the sort of automation scenarios that VBA users were accustomed to. Moreover, the API design itself was limiting, as there were almost no backing objects to represent the richness of the Office document. The omission was no accident: a rich object model implies objects that have countless properties and methods, but making each of them an async call would have been not only cumbersome to use, but also highly inefficient. The user would still be waiting their 2.5 seconds or 25 seconds, or 4+ minutes for the operation to complete, albeit without having their browser window frozen.

The new **Office 2016 API model** offers a radical departure from the Office 2013 design. The object model – now under the `Excel` namespace for Excel, `Word` for Word, `OneNote` for OneNote, etc., – is backed by strongly-typed object-oriented classes, with similar methods and properties to what you'd see in VBA. Interaction with the properties or methods is also simple and sequential, similar in spirit to what you'd do in VBA or VSTO code.

Whoa! How is this possible? The catch is that, underneath the covers, setting properties or methods **adds them to a queue of pending changes, but *doesn't dispatch them* until an explicit `.sync()` request. That is, the `.sync()` call is the *only* asynchrony in the whole system**. When this `sync()` method is called, any queued-up changes are dispatched to the document, and any data that was requested to be loaded is received and injected into the objects that requested it. Take a look at this incarnation of the cell-highlighting scenario, this time written using the new Office.js paradigm, in JavaScript:

Office.js incarnation of the VBA macro, with blocks of still-seemingly-synchronous code

```
Excel.run(function (context) {
    var selectionRange = context.workbook.getSelectedRange();
    selectionRange.format.fill.clear();

    selectionRange.load("values");

    return context.sync()
        .then(function () {
            var rowCount = selectionRange.values.length;
            var columnCount = selectionRange.values[0].length;
            for (var row = 0; row < rowCount; row++) {
                for (var column = 0; column < columnCount; column ++) {
                    if (selectionRange.values[row][column] > 50) {
                        selectionRange.getCell(row, column)
                            .format.fill.color = "yellow";
                    }
                }
            }
        })
        .then(context.sync);

}).catch(OfficeHelpers.Utilities.log);
```

As you can see, the code is pretty straightforward to read. Sure, there are the unfamiliar concepts of `load` and `sync`, but if you squint over the `load` and `sync` statements and the `Excel.run` wrapper (i.e., only look at lines **#2-3**, and then **#9-18**), **you still have seemingly-synchronous code with a familiar-looking object model**.

If instead of plain JavaScript you use TypeScript (see *A word on JavaScript and TypeScript*), the Office.js code becomes even cleaner.

*The same Office.js rendition, but this time making use of **TypeScript 2.1’s ‘async/await‘** feature*

```
 1  Excel.run(async function (context) {
 2      let selectionRange = context.workbook.getSelectedRange();
 3      selectionRange.format.fill.clear();
 4
 5      selectionRange.load("values");
 6      await context.sync();
 7
 8      let rowCount = selectionRange.values.length;
 9      let columnCount = selectionRange.values[0].length;
10      for (let row = 0; row < rowCount; row++) {
11          for (let column = 0; column < columnCount; column ++) {
12              if (selectionRange.values[row][column] > 50) {
13                  selectionRange.getCell(row, column)
14                      .format.fill.color = "yellow";
15              }
16          }
17      }
18
19      await context.sync();
20
21  }).catch(OfficeHelpers.Utilities.log);
```

In fact, if you ignore the `Excel.run` wrapper code (the first and last lines), and if you squint over the `load` and `sync` statements (lines **#5-6** and **#18**), the code looks reasonably similar to what you’d expect to write in VBA or VSTO!

Still, programming using the new Office.js model – and, for VBA or VSTO users, adapting to some of the differences – has a definite learning curve. This book will guide you through getting started, understanding the API basics, learning the unconventional tricks of debugging Office.js code, and grasping the key concepts for writing performant and reliable add-ins. It will also give tips on making your development experience easier, on utilizing some of the lesser-known functions of Office.js framework, and on writing testable code. Finally, it will address some frequently asked questions, and give guidance on broader topics such as how to call external services, how to authenticate within Office add-ins, and how to publish and license your Add-ins.

2.5.2 What is meant by "the server"

Throughout the Office documentation, as well as this book, you may occasionally encounter sentences such as "after calling `chart.getImage()` and doing `context.sync()`, the server will populate the result object with the requested image". What is meant by "**the server**" when you're issuing a simple API request to Excel? Does this mean that even when you're working on the Desktop, something is being processed by some remote Office 365 endpoint?

Don't let the wording confuse you. In the context of an Office Add-in, by *the server*, what is really meant is *the host application* – that is, the Office application that contains your add-in, and which ultimately manipulates the document on the add-ins behalf. In the case of Office Online, this is in fact a remote Office 365 server. In case of Office on the Desktop or Mac or on iOS, however, it's just the Excel/Word/etc. application. So, in all cases except Office Online, the operation is executed locally – no remote back-end server crunches the request. The reason it's sometimes referred to as *the server* is that, from the running JavaScript's perspective, it's still a remote and unknown endpoint, somewhere beyond the JavaScript runtime's application boundary… And so it's still a "server" from that perspective, even if it's often just a different process on the same machine.

3. Getting started: Prerequisites & resources

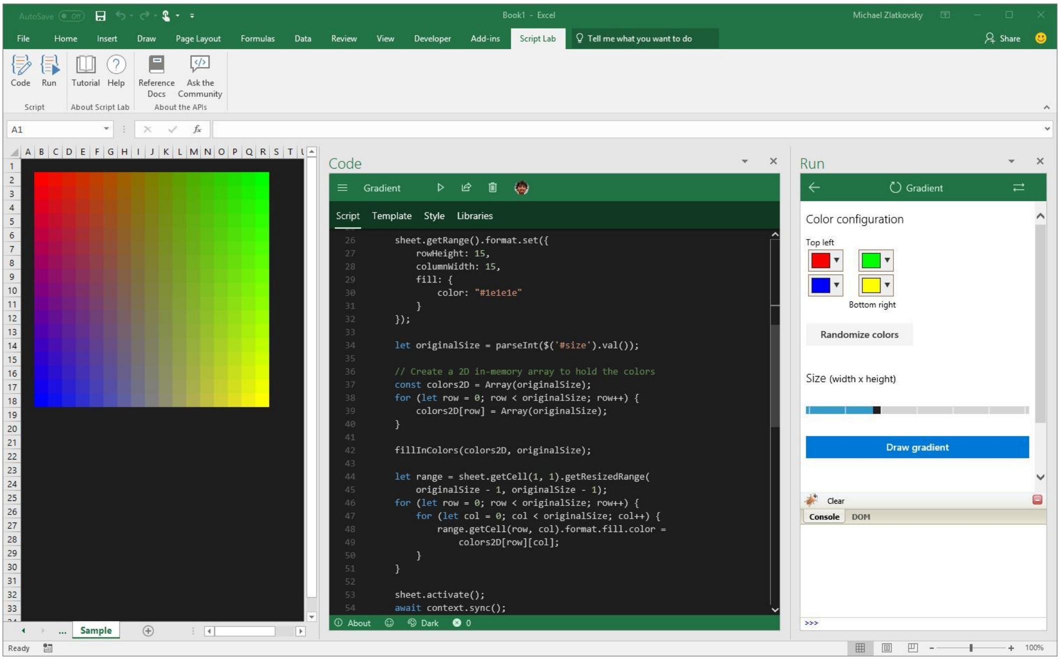
This chapter will focus on *the minimal prerequisites* for getting started with *learning* the Office.js APIs. The goal of this chapter is to quickly get you started on *being able to try out code samples from this book*.

When you are ready to create a new project from scratch (and need guidance on how to enable TypeScript, for instance), or when you need to learn about different Office versions, API versioning, and other practical aspects of building an add-in, please see a later chapter, "*The practical aspects of building an Add-in*".

Chapter structure

3.1 Script Lab: an indispensable tool

For any and all samples in this book, I *highly* encourage you to try them out in Script Lab – a coding playground where you can experiment with Office.js APIs straight from within Excel, Word, and etc. Script Lab, built itself as an Office Add-in, is available for free from the Office Store: https://aka.ms/getscriptlab.



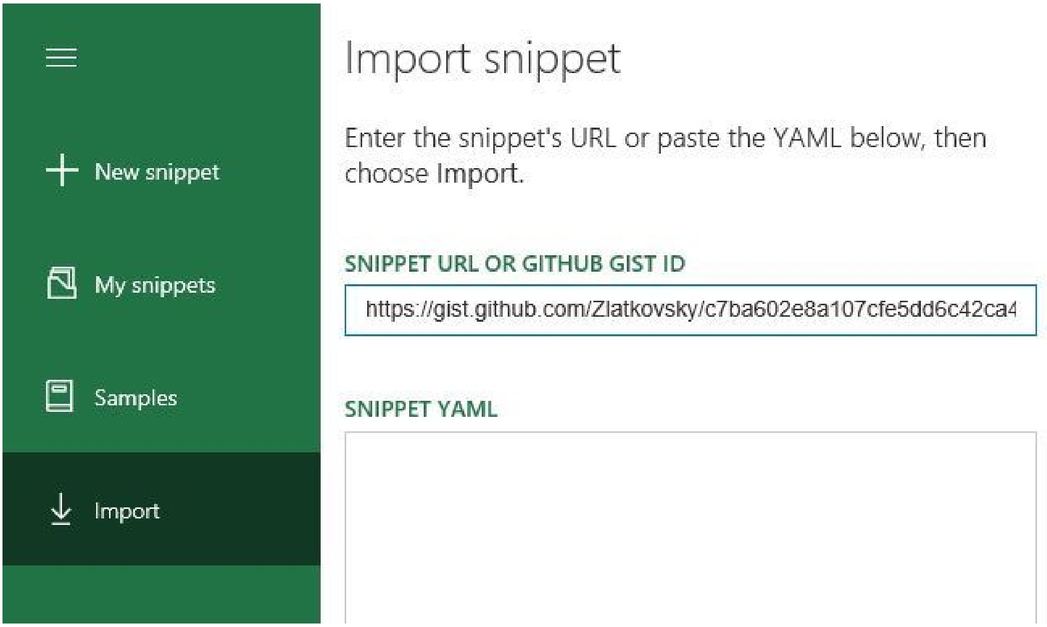
```
    sheet.getRange().format.set({
        rowHeight: 15,
        columnWidth: 15,
        fill: {
            color: "#1e1e1e"
        }
    });

    let originalSize = parseInt($('#size').val());

    // Create a 2D in-memory array to hold the colors
    const colors2D = Array(originalSize);
    for (let row = 0; row < originalSize; row++) {
        colors2D[row] = Array(originalSize);
    }

    fillInColors(colors2D, originalSize);

    let range = sheet.getCell(1, 1).getResizedRange(
        originalSize - 1, originalSize - 1);
    for (let row = 0; row < originalSize; row++) {
        for (let col = 0; col < originalSize; col++) {
            range.getCell(row, col).format.fill.color =
                colors2D[row][col];
        }
    }

    sheet.activate();
    await context.sync();
```

Script Lab in action. Get your free copy of Script Lab at https://aka.ms/getscriptlab

In addition to creating your own snippets in Script Lab, you can also *import* someone else's, or *share* yours (e.g., to post alongside a question on StackOverflow). To quote the Script Lab README.md file:

> ### Import someone else's snippet, or export your own
>
> Script Lab is built around sharing. If someone gives you a URL to a GitHub Gist, simply open Script Lab, use the hamburger menu at the top left to see the menu, and choose "Import" category (either

on the left or top, depending on the available screen space). Then, enter the URL of the Gist, and click the "Import" button at the bottom of the screen. In just these few clicks, you will be able to view and run someone else's snippet!



Import snippet

Enter the snippet's URL or paste the YAML below, then choose Import.

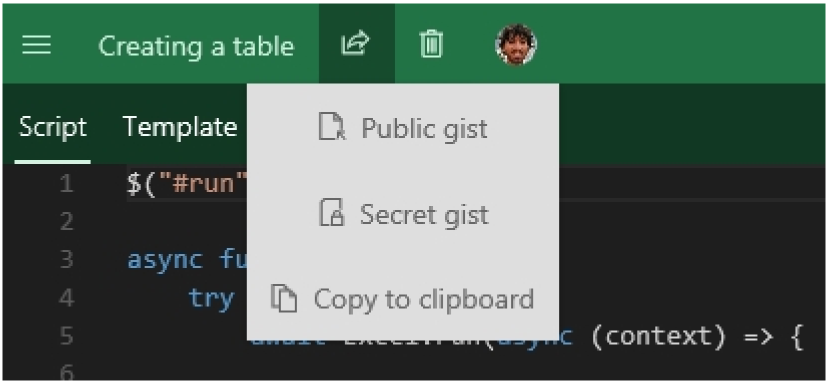
SNIPPET URL OR GITHUB GIST ID

https://gist.github.com/Zlatkovsky/c7ba602e8a107cfe5dd6c42ca4

SNIPPET YAML

Import tab in the "Hamburger" menu

Conversely, to share *your* snippet with someone, choose the "Share" menu within a particular snippet. You can share as a public or private GitHub Gist, or you can copy the entire snippet metadata to the clipboard, and share it from there.



```
$("#run"

async fu
    try
        await Excel.run(async (context) => {
```

Share menu

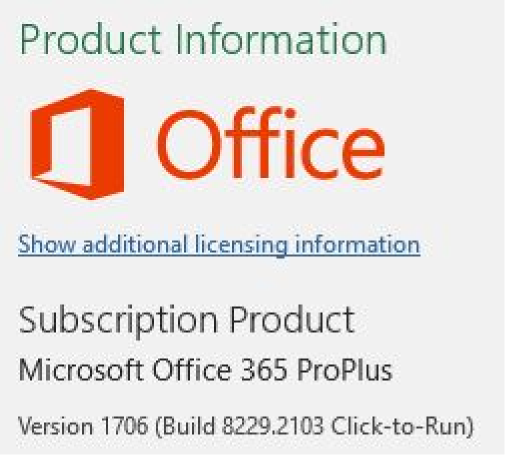
For purposes of what you need to know to use Script Lab, that’s about it! But if you’re curious about the project origins, see “*Script Lab: the story behind the project*” in Appendix B.

3.2 The optimal dev environment

For a developer environment (and what I use myself), I recommend the following:

1. A Windows machine. You can develop on other environments too, but the Windows version of Office is generally ahead of the Mac in terms of API functionality; and as for developing using Office Online as the primary environment, you will find it slower and more limiting that the Desktop versions (at least for Excel and Word). So, having a Windows machine is best.

2. The latest copy of Office, installed *via an Office 365 subscription*. The "**subscription**" piece is critical here, because this will let you get monthly updates (with new features and bug fixes for the APIs). Just an MSI install of Office 2016 is *not sufficient* if you want an optimal developer experience. See the topic on Office versions for a more in-depth explanation.



Product Information

Office

Show additional licensing information

Subscription Product

Microsoft Office 365 ProPlus

Version 1706 (Build 8229.2103 Click-to-Run)

You want to make sure you have the subscription version of Office

3. A copy of Script Lab. As mentioned in the previous section, Script Lab will make it *so much easier* for you to try out the code exercises, as well as explore on your own. It comes already pre-configured with TypeScript, IntelliSense that works right-out-of-the-box, the ability to share and import, and more. If you haven't already, get the Script Lab add-in free from the Store: https://aka.ms/getscriptlab.

Realistically, that's all you need to get started with learning the APIs. However, for completeness sake, let me also add a few more tools that you'll want to have, once you start developing a real add-in:

4. Visual Studio. Even if you don't intend to use Visual Studio for creating/editing the projects (as opposed to using a combo like Node + NPM + VS Code, which I describe later) – you'll still need Visual Studio on your machine in order to **debug** your code (except for testing in Office Online, where you can just use the browser's F12 Developer Tools). See "*Debugging: the bare basics*" for more information on either option.

5. [Optional] Node + NPM + Visual Studio Code. Finally, if you are interested in using modern web tooling, you will need Node and NPM installed (https://nodejs.org/download). I would also recommend Visual Studio Code as an excellent editor that has great IntelliSense (auto-completion) support, a built-in terminal, a bunch of extensions, and more.

If instead of practicing in Script Lab, you'd prefer to practice within your own project, please see the "*The practical aspects of building an Add-in*" for step-by-step instructions on creating a project. Otherwise, just use Script Lab for now, and you can switch over later!

3.3 API documentation resources

The official documentation for Office.js can be found at https://dev.office.com.

The documentation is especially helpful for details about specific APIs, since this book focuses on the exact opposite: the overall patterns and practices of the APIs, but not on any individual object or method.

Some top-level links about the different API surface areas are as follows:

- **Excel:** https://dev.office.com/docs/add-ins/excel/excel-add-ins-javascript-programming-overview
- **Word:** https://dev.office.com/reference/add-ins/word/word-add-ins-reference-overview
- **OneNote:** https://dev.office.com/docs/add-ins/onenote/onenote-add-ins-programming-overview
- **The 2013 wave of "Common" APIs:**
  - Word & Excel: https://dev.office.com/reference/add-ins/javascript-api-for-office
  - PowerPoint: https://dev.office.com/docs/add-ins/powerpoint/powerpoint-add-ins
  - Project: https://dev.office.com/docs/add-ins/project/project-add-ins
- **Outlook** (which still uses the Office 2013 syntax – so not as topical for this book – continues to expand its API surface area to new Outlook-specific scenarios): https://docs.microsoft.com/en-us/outlook/add-ins/

Also, for APIs that are not yet implemented or are in **beta**, you can view the Open Specs here: http://dev.office.com/reference/add-ins/openspec.

4. JavaScript & Promises primer (as pertaining to our APIs)

To those first entering JavaScript from the traditional C#/Java/C++ world, JavaScript may seem like the Wild West (and it is). While it's beyond the scope of this book to teach you JavaScript all-up, this chapter captures some important concepts in JavaScript – and particularly JavaScript's "Promises" pattern, and the wonderful way in which TypeScript's `async/await` simplifies it – that will greatly improve the quality and speed at which you write Office Add-ins.

If you're a JavaScript/TypeScript pro, you may still find some nuggets of information useful, but you can also skim/skip ahead if you want to – there is nothing Office.js-specific here. On the other hand, if you're *not*, I highly recommend reading this chapter and probably returning to it a number of times later for reference. JavaScript concepts, and especially Promises, are 100% prevalent and essential for understanding Office.js. Jumping straight ahead to Office.js concepts without understanding the JavaScript & async-programming basics, is a bit like going off a ski jump before getting your ski footing on solid ground first[1].

[1] Says the person who, in high school, effectively did just that on his first day at a ski resort… And so can speak with great authority on the subject.

Chapter structure & planned content

Note: Sections that have already been written, and are **included** in this book, are **bolded and hyperlinked**; the rest are planned topics, but have not been written (or polished) yet.

4.1 “JavaScript Garden”, an excellent JS resource

To preface this chapter, and before giving specific JS guidance of my own, let me point the reader to “JavaScript Garden”[2]. The website – essentially one very long article, broken into sections for ease of navigation – offers an excellent introduction to all things JavaScript. Per the description:

> *JavaScript Garden is a growing collection of documentation about the most quirky parts of the JavaScript programming language. It gives advice to avoid common mistakes and subtle bugs, as well as performance issues and bad practices, that non-expert JavaScript programmers may encounter on their endeavours into the depths of the language.*

The *JavaScript Garden* still assumes some familiarity with JavaScript/programming, but it starts off at a pretty basic level, and builds up the reader’s knowledge from there.

One thing to note for both the *JavaScript Garden*, and any other JS reference I’ve ever seen in my life: for some reason, folks are *fascinated* with describing how bizarre the JavaScript notion of classes and prototypes is. While I don’t disagree that the JS notion is indeed bizarre, for purposes of *just* Office.js programming, you won’t need to create JS classes – all you’ll be doing is using them, which is no different than using any other JS object. Moreover, if you decide to go with TypeScript rather than *plain ol’* JavaScript[3], you won’t even need to understand the strange `prototype` model. So, in my humble opinion, you can safely skip the section on “The Prototype” (i.e., classes) within the *JavaScript Garden* site, and any other reference that tries to bamboozle you with this topic. While classes might be helpful for compartmentalizing your logic – and while they might be essential for use in frameworks like Angular

[2] See http://javascriptgarden.info. The site is also available as a GitHub-hosted page at http://bonsaiden.github.io/JavaScript-Garden.

[3] I’m referring to JavaScript as in the JS EcmaScript 5 specification – which is the baseline of what most/all modern browsers support as of 2016. EcmaScript 6 (also known as ES6) offers a wide array of improvements, with classes being one of them. However, for wide browser support, you must use either TypeScript (its own extended-JS-syntax language), or a tool like Babel or Traceur (transpilers) in order to compile the ES6-compliant code down to the *lingua franca* that is ES5.

2, which you might use alongside Office.js – the understanding of how JavaScript classes are structured/created is not necessary for using Office.js.

For all the rest of the topics, the content found in the “JavaScript Garden” is an excellent read.

For all the cat-lovers out there, I would also be remiss not to mention “JavaScript for Cats”[4], a quirky but fun tutorial of the JavaScript language from a feline perspective.

[4] http://jsforcats.com/, also available at https://github.com/maxogden/javascript-for-cats

4.2 JavaScript & TypeScript crash-course (Office.js-tailored)

4.2.1 Variables

- Variables in plain ES5 JavaScript are declared using the **`var`** keyword: for example,

  ```
  var taxRate = 8.49;
  ```

- With few exceptions, you generally call methods or access properties *on the object itself*. This should be familiar to a C#/Java audience, but might not as much to the VBA crowd, who is used to certain methods like `mid` or `len` to be global methods that accept an object, rather than being methods *on the object*.

- Variables in *plain* JavaScript do not have an *explicit* type. This means that you always declare something as `var` (or `let`). You do **not** specify the type during declaration (i.e., there is no "`dim taxRate as Integer`" or "`int taxRate`", as you'd see in VBA or C#, respectively).

- The primitive types are just:

  - Booleans: "`var keepGoing = true`" (or `false`).
  - Numbers. There is no distinction between integers and decimals ("`var count = 5`" versus "`var price = 7.99`"), they're all numbers.
  - Strings, discussed separately in a different section.

4.2.2 Variables & TypeScript

- TypeScript offers an additional keyword, "**`let`**", which can be used in place of `var` and has cleaner semantics in terms of scoping rules (i.e., variables declared inside a loop using `let` will only be visible/usable in that loop, just as you'd expect with VBA or C#, but *not* what you get with just the JavaScript `var`). Because of this, when using TypeScript, it is **always a best-practice to use "`let`"**. In fact, if you're only assigning a variable once and never re-assinging it, it is better to use an even stronger form of `let`: "**`const`**".

- TypeScript, true to its name, let you ***optionally*** (and gradually) add type information to variables, which you do by appending a colon and a type name after the declaration (you'll see examples momentarily). Common types include:

  - Primitive types, such as `number`, `string`, `boolean`.
  - Array of types. This can be written as "`number[]`" or "`Array<number>`", it's a stylistic preference.
  - Complex types (esp. ones that come with the Office.js object model). For example, "`Excel.Range`" or "`Word.ParagraphCollection`".

- Specifying types is generally used for:

  - Variable declarations, especially if it's necessary to declare a variable in advance, before assignment:

    ```
    let taxRate: number;
    ```

  - Parameter type declarations. This is supremely useful, as it allows you to split long procedures into multiple subroutines, and yet not lose IntelliSense and compile-time error-checking. For example, you can declare a function as

    ```
    function updateValues(range: Excel.Range) { ... }
    ```

    and be able to see all of the `range` properties/methods/etc. within the body of the function, as opposed to seeing a dishearteningly-empty IntelliSense list. See the *"IntelliSense across functions"* image in section *"A word on JavaScript and TypeScript*"*.

  - Function return-type declarations, such as

    ```
    function retrieveTaxRate(): number { ... }
    ```

    That way, when assigning the value to a variable, the variable will get implicitly typed. Thus,

    ```
    let taxRate = retrieveTaxRate();
    ```

    will now give you the full IntelliSense for a number, rather than an empty list for an unknown (“`any`”) type.

- Very occasionally, you might have disagreement with the TypeScript compiler over a particular type. This happens primarily in Excel for 2D-array properties, such as `values` or `formulas` on an `Excel.Range` object. For those properties, the *return value* is always a 2D array, but they are more accepting in terms of their *inputs* (namely, they can accept a single non-arrayed value). TypeScript, however, does not recognize such subtleties.

  Fortunately, TypeScript offers a simple escape hatch from its type system: just stick an “**`<any>`**” in front of the value you’re setting, and TypeScript will let the issue go. So, if you want to set a single value to a property that’s technically a 2D-array, just do something like

  ```
  range.values = <any> 5;
  ```

  and both you and the TypeScript compiler will walk away happy.

4.2.3 Strings

- Strings can be either be single-quoted or double-quoted, so long as the quote types match. Strings can also be concatenated using the "+" sign. The following is 100% legal:

  ```
  var message = 'hello ' + "world";
  ```

- If you find yourself in need of both single and double quotes (*"it's terrible", said she*), you can use the backslash ("\") symbol to escape the quote, as in:

  ```
  console.log('"It\'s terrible", said she');
  ```

- Use "`.trim()`" to remove whitespaces before or after the text. Particularly useful when accepting user input.

- To convert a string to a number, use

  ```
  parseFloat(text);
  ```

  or

  ```
  parseInt(text);
  ```

- Use "`.substr`" or "`.substring`" on the string object to extract a particular portion of the string. This is often used in conjunction with an "`.indexOf`". See http://stackoverflow.com/a/3745518/678505 for the difference between "`.substr`" and "`.substring`".

- Unlike most languages, the "`.replace`" function of a string only replaces the first match. See http://stackoverflow.com/a/17606289/678505 for a good workaround.

- Strings work really well in conjunction with Regular Expressions for extracting/replacing data. See http://eloquentjavascript.net/09_regexp.html for one of many good tutorials.

- **TypeScript** offers a wonderful way to concatenate strings using "*template strings*". See https://basarat.gitbooks.io/typescript/content/docs/template-strings.html

4.2.4 Assignments, comparisons, and logical operators

- To assign a value to a variable, use a single equals sign, as in
  ```
  var found = false;
  ```

- To compare whether two values are equals, use a double-equals or triple-equals (slightly different semantics, with the latter being a strict equality, as opposed to the loosey-goosey concept of truthy/falsy values). Best-practice is to use triple-equals:
  ```
  if (count === 2) { ... }
  ```

- Other comparison operators include "`!=`" for "NOT equal" (with its stricter best-practice cousin, "`!==`"), as well as ">", ">=", "<", and "<=".

- You can use the exclamation mark (`!`) without the equals sign to negate an entire statement – for example,
  ```
  if (!keepGoing) { ... }
  ```

- To "AND" statements together, use a double-ampersand:
  ```
  if (values.length > 0 && values[0].length > 2) { ... }
  ```
  To "OR" statements, use a double-pipe:
  ```
  if (name === "" || price === "") { ... }
  ```

4.2.5 `if`, `for`, `while`

- As seen earlier, an `if` statement consists of the word "`if`", followed by a conditional statement within the parentheses, and then the body of the function (what to do on success) inside of curly braces. For example:

```
if (!keepGoing) {
    console.log("Quitting");
}
```

- If you want an "`else`" or "`else if`" statement, the structure is as follows:

```
if (x === 0) {
    ...
} else if (x === 1) {
    ...
} else {
    ...
}
```

- If you end up with a bunch of "`else if`" statements, consider using a "`switch`" statement, instead (see http://www.w3schools.com/js/js_switch.asp).

- JavaScript has a curious concept of truthy/falsy values, where you can essentially put *anything* into an "`if`" statement – and so long as it's not `null`, `undefined`, 0, `false`, or an empty string, it will be evaluated as `true`.

- A `for` loop looks as follows:

```
for (var i = 0; i < 5; i++) {
    ...
}
```

  Note that "`++`" is simply shorthand for "`i = i + 1`". This means that in the context of the `for` loop, this means "*increment* `i` *by 1 with each iteration of the loop*". You can specify other increments as well.

- A `while` loop looks as follows:

```
while (keepGoing) {
```

```
                ...
            }
```

4.2.6 Arrays

- To declare an empty array, simply assign “`[]`” to the array – that is,
  ```
  var products = [];
  ```

- To declare a *non*-empty array, just assign values inside of the square brackets. For example,
  ```
  let products = ['Kiwi', 'Strawberries'];
  ```

- To add new objects onto the array, use the `.push` method. For example,
  ```
  products.push('Raspberries');
  ```
  Other useful methods are `.pop` (remove the last element of the array) and `.shift` (remove the first).

- To get a count of elements, use the “`.length`” property:
  ```
  console.log(products.length);
  ```

- To retrieve a particular element by index, use the square-bracket syntax:
  ```
  var firstProduct = products[0];
  ```

- To iterate through an array, you can do a regular `for` loop, as follows:
  ```
  for (var i = 0; i < products.length; i++) {
      console.log(products[i]);
  }
  ```

- You can also iterate using a “`.forEach`” function, as in:
  ```
  products.forEach(function(item) {
      console.log(item);
  });
  ```

- There are a variety of other methods that accept functions, similarly to “`.forEach`”. Most notable are “`.filter`”, “`.sort`”, “`.indexOf`”, and “`.map`” (the latter performs a transformation on each element of the object; very powerful).

- ”`.join`” is another very handy function, to join together elements of the array into a string (for example, comma-separated list of items)

- For a 2D array (which is quite common in Excel APIs, for range values, formulas, number-formats, etc – and which always have the outer array elements representing rows, and the inner one columns):

  - You can access an individual items via two square brackets one after another. For example, `values[2, 0]` will give the element at relative row index 2 (third row) and column index 0 (first column).

  - You can get the count of rows via `values.length`. To get a count of columns, grab the first row, and ask *it* for its count: `values[0].length`.

  - To iterate through a 2D array, do:

    ```
    for (var row = 0; row < values.length; row++) {
        for (var col = 0; col < values[0].length; col++) {
            var element = values[row][column];
            console.log(element);
        }
    }
    ```

  - You can create a 2D array from static data, such as:

    ```
    var values = [
        ["Product", "Price"],
        ["Kiwi", 3.99],
        ["Apple", 1.76]
    ];
    ```

  - To create a 2D array dynamically, it's easiest to create an empty outer array. Then, create new arrays that represent a single row's data, and push this newly-created array onto the outer array. For example, to create the values inside of a multiplication table:

    ```
    var data = [];
    for (var row = 1; row < 10; row++) {
        var singleRowData = [];
        for (var col = 1; col < 10; col++) {
            singleRowData.push(row * col);
        }
     data.push(singleRowData);
    ```

```
                }
```

4.2.7 Complex objects & JSON

- One of JavaScript's greatest strengths is the ease of creating data structures. You simply create an empty object using "`{}`" and then populate it later, or you create it with data from the start, using key-value pairs that are separated by a colon between the key and the value, and commas between pairs. Thus, to create a data structure that contains a product's name and price, simply do:

  ```
  var productInfo = { name: 'Kiwi', price: 3.99 );
  ```

- Reading data back from an object uses square-bracket notation

  ```
  var priceForTwo = productInfo['price'] * 2;
  ```

- Adding (or updating) a property also uses square-bracket notation, but this time with an assignment. For example, to update the price:

  ```
  productInfo['price'] = 3.50;
  ```

  Adding a brand-new property is identical:

  ```
  productInfo['quantity'] = 6;
  ```

- The objects can be deeply-nested, and can also contain arrays. For example:

  ```
  var cellData = {
      values: [
          ["Product", "Price"],
          ["Kiwi", 3.99],
          ["Apple", 1.76]
      ],
      format: {
          fill: {
              color: "green"
          }
      }
  ];
  ```

- To create a dictionary of key-value pairs, just create a single empty object ("`{}`"), and populate it with properties that are *keys* to the data you want. For example:

  ```
  var inventory = {
  ```

```
        kiwi: { price: 3.99, quantity: 6 },
        apple: {price: 1.76, quantity: 13 }
    };
```

Note that the property names could also be quoted: that is, `"kiwi"` and `kiwi` are identical. BUT, if you want a property name that contains whitespaces or other non-alpha-numeric characters in it, those do need to be quoted:

```
    var inventory = {
        kiwi: { price: 3.99, quantity: 6 },
        "granny smith apple": { price: 2.00, quantity: 20 }
    };
```

- To serialize objects into a string (e.g., for storing into a setting), you can do

  ```
      JSON.stringify(productInfo);
  ```

  Likewise, to de-serialize a previously-serialize object (or data sent over the wire), use

  ```
      JSON.parse(stringifiedData);
  ```

4.2.8 Functions

- The simplest possible function is declared as follows:
  ```
  function doSomething() { ... }
  ```

- More interesting functions take in some parameters. For example:
  ```
  function formatParagraph(paragraph, indent) { ... }
  ```
  Note that there is neither a "`var`" nor a type in front of parameters, you simply list them out in order.

- JavaScript functions (though not with TypeScript!) are very loose when it comes to parameter counts: you can pass in more parameters than the function accepts, or less, and the runtime won't complain in either case. Extra parameters will simply be ignored, and insufficient paramters will be seen as having an `undefined` value.
- The order between the function declaration and where it's used does *not* matter. You can invoke a function even if it is technically declared somewhere below in the source code.

- Functions are "first class citizens" in JavaScript. This means that functions can (and very often are) passed around between methods, especially as parameters to *other functions*. For example, you might pass in a function name to a click handler, as in
  ```
  $('#setup).click(doSomething);
  ```

- Quote often, instead of passing around "named" functions (as above), you'll see anonymous functions being used. For example,
  ```
  $('#setup).click(function() {
      console.log("I was clicked!");
  });
  ```

  Another example is the "`array.forEach`" above. And, as you'll see in upcoming chapters, the `Excel.run`, `Word.run`, etc. methods all take in a function that describes what object-model manipulations you'd like to do with the document.
- Unlike VBA/C#/etc., JavaScript allows (and encourages) the creation of *nested-functions*: that is, functions within functions. For example, imagine you have a function called `createReport`, which does a bunch of op-

erations. If you want to break up the function into smaller subroutines, you can choose to place them either outside (as you would in VBA/C#) *or inside* the function body. The purpose of putting functions inside is generally for encapsulation, to make it clear that a given method is only meant to be used in-context (i.e., may not be multi-purpose), and to avoid name collisions (in ES5 and earlier, without proper support for classes, functions often served the role of encapsulation). Functions that are nested within another function will have access to all of the local variables and parameters that are passed into the function, which can also be quite convenient.

4.2.9 Functions & TypeScript

As with variables and types, TypeScript brings a couple of enhancements to functions.

- TypeScript function declarations can declare parameter types. For example:

  ```
  function formatSheet(sheet: Excel.Worksheet) { ... }
  ```

  Optional paramters can be included at the end, with a `?` syntax:

  ```
      function formatSheet(sheet: Excel.Worksheet, color?: string) { ...
  }
  ```

  You can also default the optional parameter to a particular value by removing the question mark, and instead setting the value after the type:

  ```
      function formatSheet(sheet: Excel.Worksheet, color: string = "white")
  { ... }
  ```

- As with incoming parameters, a function can also declare its return type. For example a function that returns an Excel Range object might be denoted as:

  ```
  function insertRow(sheet: Excel.Worksheet): Excel.Range { ... }
  ```

- TypeScript offers an alternate "fat-arrow" syntax[^fat-arrow] for anonymous function declarations, as in:

  ```
  $('#setup).click(() =>
      console.log("I was clicked!");
  });
  ```

  Yes, "*fat arrow*" is a real term, and you have to admit it's memorable! Importantly, when working with classes, the *fat-arrow* syntax also captures the `this` variable, making it a "best practice" to use anywhere when you're using TypeScript (much as "`let`" is an improved "`var`", even if the distinction isn't always necessary). It is also shorter to type.

- Related to the above: the “fat arrow” syntax can also provide an implied `return` statement, when there is only a single statement being made (and returned out) in the function body. For example,

  ```
  array.map((item) => item.trim());
  ```

  is identical to the more verbose:

  ```
  array.map(function (item) {
      return item.trim();
  );
  ```

- TypeScript also offers a way to create *function overloads* (functions by the same name, which accept differing parameters). See https://www.typescriptlang.org/docs/handbook/functions.html, and search for “Overloads”.

4.2.10 Scope, closure, and avoiding polluting the global namespace

- Scoping rules in JavaScript are unusual, in that *only functions* create a new scope, but not constructions like `if` statements, `for` loops, etc. They are *particularly unusual* when looping over a variable and attaching a deferred action. For example, if I create 5 buttons using a `for` loop, and expected each to output its index number I clicked, this code – which uses **`var`**, would be ***incorrect***:

  ```
  for (var i = 0; i < 5; i++) {
      var $button = $('<button>').text(i).appendTo('#buttons');
      $button.click(function() { console.log(i); });
  }
  ```

  When run, the code would always output “`5`”, because the scope of “`i`” isn’t captured by the click handler. **My recommendation: use TypeScript and the `let` keyword, which avoid this problem altogether**. Or, if you have no choice, use a JS workaround[5].

- To avoid polluting the global namespace, it is best to declare all variables inside of some non-global scope, and only have a single entry-point (“`MyApp`”) that exposes functions. **TypeScript offers a very simple way of doing this, using *namespaces***. Namespaces also offer you the option to keep some variables completely private, while allowing others to be exposed via the namespace:

  ```
  namespace MyApp {
      var somethingPrivate = 5;
      export var somethingPublic = somethingPrivate * 2;
  }

  console.log(MyApp.somethingPublic);
  ```

  In the absence of TypeScript, you can still create a scope yourself, using an anonymous self-executing function – which is not necessarily hard,

[5] Search for the words “*anonymous closure*” within https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures.

but looks bizarre. Coincidentally, this is pretty close to what TypeScript itself would generate for a namespace[6].

```
var MyApp = {};
(function () {
    var somethingPrivate = 5;
    MyApp.somethingPublic = somethingPrivate * 2;
})();
```

[6] Close, but simplified relative to what TypeScript itself outputs. See it for yourself at http://www.typescriptlang.org/play/index.html.

4.2.11 Misc.

- To help the JS runtime help *you* catch errors (e.g., using an undeclared variable, setting a read-only property, etc.), include a `"use strict"` (quoted, just like that) at the top of your file, before anything else. More info here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode
- JavaScript statements are (best) separated by semicolons. JS will often guess correctly *even if you omit the semicolon*, but things may break down unexpectedly under certain situations. Best practice is to always include semicolons at the completion of each statement.

- Comments are same as in C/C++/C#/Java.
  For single-line comments, you can use
  ```
  // <your comment text>
  ```
  (where anything after the "`//`" is treated as a comment).
  For multi-line comments, or to control the start and end location, use
  ```
  /* <comment text */
  ```

- Error-handling is done via `try/catch` blocks for normal *synchronous* code, or via a `.catch` method for asynchronous Promises. You'll read about both more later in the chapter.

- When debugging, it is often useful to write out to the console (provided that you have a debugger attached, and are watching the JavaScript console output). To write to the console, simply do
  ```
  console.log('whatever you want to log');
  ```

  You can also pass objects to the `console`. Sometimes, a simple
  `console.log(theObject)`
  works best, and allows you to drill in to object properties; at other times, JSON-stringifying the object works better.

- JavaScript object-passing semantics are pretty simple, and are wonderfully explained in
  http://stackoverflow.com/a/6605700/678505 by @nrabinowitz.
  Quoting from that answer (see the answer itself for some code examples):

- *Javascript is always pass by value, but when a variable refers to an object (including arrays), the "value" is a reference to the object.*
- *Changing the value of a variable never changes the underlying primitive or object, it just points the variable to a new primitive or object.*
- *However, changing a property of an object referenced by a variable does change the underlying object.*

4.2.12 jQuery

jQuery is one of the original libraries to help create and wire-up web UI in a sane way. jQuery would hide some of the behavioral differences between browsers, exposing a convenient syntax for *manual UI creation*

I say *manual* because there are a variety of other frameworks – Angular or React, to name some recently-popular ones – that are much more magical, but also a lot more complex. This is akin to the difference between WinForms and WPF, for those familiar with creating UI for Windows desktop. jQuery will allow you to easily wire-up button-click events, and to manually show/hide elements of the UI – but it won't do it for you. Angular and friends, on the other hand, will have you define a *model* and specialized markup that get projected into the UI, complete with two-way bindings and all that jazz… but at the cost of complexity.

Personally, especially for quick prototyping of API functionality, I just use jQuery. In a few of the prior examples, when you saw the "`$`" symbol – that was jQuery. You use `$(selector)` to retrieve (or create) UI elements, and then call functions on it to pass in data. The selectors are either class-names (prefaced with a dot) or ids (prefaced with a "`#`" sign) that come from the corresponding HTML.

So, given some basic HTML:

```
...
<body>
    <button id="show-time">Show time</button>
    <div id="time"></div>
</body>
...
```

You could do:

```
$('#show-time').click(updateTime);

function updateTime() {
    $('#time').text(new Date().toTimeString());
}
```

Other common actions include calling “`.show()`” or “`.hide()`” on an element (or elements, if retrieving multiple via a class name), appending new elements (as in one of the examples earlier), and adding or removing classes (which, in turn, would correspond to CSS styling). jQuery also offers a “`$.ajax(...)`” method, for retrieving data from the internet – you will see an example of it in a later section.

There are countless tutorials available on the internet for jQuery, such as these quick-getting-started ones:

- http://blog.webagesolutions.com/archives/138
- https://learn.jquery.com/about-jquery/how-jquery-works/

4.3 Promises Primer

Most modern web applications rely on some form of asynchronous operations, for things like fetching data from a web service, or pushing data to the cloud. Until not long ago (∼2014/2015), it was common-practice for developers to rely on *callback functions* for resuming operations at the completion of a particular async call. Callbacks solved the problem, but they often resulted in deeply-nested difficult-to-read code, colloquially known as "*pyramids of doom*"[7].

The JavaScript *Promises* pattern was created to solve the async-chaining problem in a cleaner way. At their core, Promises provide a way for chaining together multiple actions, in an easy-to-read and predictable order. TypeScript 2.1 further expands upon Promises, adding an `async/await` syntax sugar on top of the the underlying Promises implementation, to make Promises a lot easier to work with. For those coming from a C# background, you'll find the TypeScript `async/await` pattern very familiar; and much like C#'s `async/await` leveraged Tasks, so does TypeScript's `async/await` leverage Promises.

[7] See an explanation on https://github.com/kriskowal/q, or just look up this term on the web. And *yes*, it is in fact a real term, for which even Wikipedia has an article: https://en.wikipedia.org/wiki/Pyramid_of_doom_(programming).

4.3.1 Chaining Promises, the right way

To the extent that Promises are chained, the term "*Breaking the Promise chain*" refers to accidentally letting some of operations run loose. When the Promise chain is broken, instead of a single strand of operations, you accidentally end up with multiple parallel execution. This is bad because:

- More often than not, logic that was meant to run in a certain order will break if the operations start running *out of order*. This leads to a class of not-always-reproducible and hard-to-diagnose bugs, known as *timing issues*.
- In the case of Office.js APIs and the `Excel.run(batch)` (and `Word.run(batch)`, etc.) family of methods, a broken Promise chain can lead to objects being inadvertently cleaned up[8] before you're done using them… leading to even harder-to-diagnose timing issues.
- A broken Promise chain can leads to silently-swallowed errors, with the `.catch` statement handling only one of the thread chains, but not the other(s).

The good news is that **in the *new* Office.js[9], the only asynchronous operation is "`context.sync()`"**. So – other than places in your code where you explicitly call a web service or do some other async operations – the only place where you need to watch out for with Office.js is just the `context.sync()` method. And moreover, **there is exactly one simple rule you need to follow:**:

[8] See "*What does `.run` do, and why do I need it?*", for details on what gets cleaned up.

[9] In the original 2013 Office.js, *Callback* were used instead of *Promises*, so the *Promise* discussion is moot for those. Only the Office 2016 wave of document-manipulation APIs makes use of Promises.

THE GOLDEN RULE OF PROMISES:

Unless you are intentionally introducing an independent strand of execution (such as an action that fires after X number of seconds, or creating a button-click listener on a DOM object), you should:

TypeScript version: Ensure that you **`await`** the completion of the Promise before moving on to subsequent execution.

JavaScript version: If done as part of chaining a previous Promise (e.g., as part of a `.then` function), ensure that the code inside the `.then` is either *either be fully synchronous* or ***returns a Promise***.

The part about `await`-in (or `return`-ing) a Promise is critical. Just calling `context.sync()` inside of the body of a function, without awaiting/returning it, will lead to anything from errors being silently swallowed, to everything going *completely* awry.

TIP: Not all promises are created equal! Promises are a fairly new concept (with Chrome and Firefox only adding native support in late 2015, and with Internet Explorer implementing Promises on Edge but not IE 11). So especially for Promise polyfills (i.e., libraries that emulate Promise-like behavior on browsers that don't natively support Promises), the Promise behaviors may vary depending on what polyfill library you are using.

An early adopter of promises was jQuery, with their Deferred object concept. However, the old jQuery implementation was not up to the newer A+/Promises spec, and may not play nice (e.g., propagate errors, etc.) when used in conjunction with native promises or with the native-like behavior of the `OfficeExtension.Promise` object. Even the jQuery 3.1+ version – while now compliant to the A+/Promises spec – still differs from standard Promise implementations (i.e., it does not have a `.catch` function, instead using its own `.fail`). My recommendation for jQuery is to be sure that you're using the latest 3.1+ version *and also* to wrap jQuery Promises in more standard promises (either manually, or through a library like Q, available at https://github.com/kriskowal/q; I've used Q many times before, and it's an excellent Promise library in its own right).

4.3.2 Creating a new Promise

The preceding section explained how to properly chain and use Promises – which is the absolute basics of what you need to know to use Office.js. However, you may find that sometimes you need to create a brand new Promise of your own. How and why would you do this?

Native promises, vs. polyfills, vs. OfficeExtension.Promise.

While you don't need anything special to *use* Promises, *creating your own Promises* does require a preparation step. Promises have only recieved mainstream browser adoption in late 2015, and so Internet Explorer 11 and earlier do *not* offer native support for them. Instead, you need to do **one of the following***, presented here in the order of ascending difficulty:

1. If you're using `ExcelApi 1.2` or `WordApi 1.2` or higher, simply use **OfficeExtension.Promise** in place of the global Promise object. This is the easiest thing to do, as Office.js includes its own Promise polyfill[a]. The examples below will use `OfficeExtension.Promise`, as it's the easiest way to get started.

2. Alternatively, use a library like Q (https://github.com/kriskowal/q), which also offers Promise-like behavior, and – like Office.js – does *not* polyfill the global `window.Promise` object. In this case, whenever you see "`new OfficeExtension.Promise(...)`" in the examples below, substitute it with "`Q.Promise(...)`" (and note that the "`new`" keyword is *not* needed for Q).

3. Include a Promise polyfill that *does* patch the global `window.Promise` object. An example of one such library is "core-js", available at https://github.com/zloirock/core-js.

[a] A *polyfill* is something that emulates native browser behavior on browsers that doesn't support a particular bit of functionality (in this case, Promises). Office.js includes a tweaked version of the ES6-Polyfill library, taken from https://github.com/jakearchibald/es6-promise. Unlike the original library, the Office.js version does *not* polyfill the global `window` object (since that may not have been the developer's intent), and instead only adds `Promise` to the `OfficeExtension` namespace, when needed. If Promised are already supported natively by the browser (as they might

for Office Online, on Edge or Firefox or Chrome), `OfficeExtension.Promise` simply redirects to `window.Promise`.

With the technical prerequisites out of the way: Let's start with *why would you want to create a Promise in the first place*. Generally, this will be when you have an asynchronous method that uses callbacks, and which you'd like to *wrap* with a Promise facade, so that it can fit in with the rest of your elegant workflow.

A very simple HTML example: Let's say that you want to take a particular object, and switch its background to reflect a color of the rainbow for a couple of seconds, before switching to the next one.

In JavaScript, you have `setTimeout`, which is a function that takes in an action and invokes it after X many milliseconds. This means that if you really wanted to, you could do the rainbow-switcher as follows:

A pyramid of ‘setTimeout‘-s

```
function createRainbow() {
    var $rainbow = $("#rainbow");
    $rainbow.css("background", "red");
    setTimeout(function() {
        $rainbow.css("background", "orange");
        setTimeout(function() {
            $rainbow.css("background", "yellow");
            setTimeout(function() {
                $rainbow.css("background", "green");
                setTimeout(function() {
                    $rainbow.css("background", "blue");
                    setTimeout(function() {
                        $rainbow.css("background", "indigo");
                        setTimeout(function() {
                            $rainbow.css("background", "violet");
                        }, 2000);
                    }, 2000);
                }, 2000);
            }, 2000);
        }, 2000);
    }, 2000);
}
```

Needless to say, this pyramid is less than ideal, as it becomes visually difficult to match the beginning and end of each function. Adding a new step in the pyramid, somewhere in the middle, also becomes tedious, as you need to carefully examine the code for the right injection point, and you need to indent all the rest of the impacted code. Finally, while not relevant for this example (`setTimeout` never errors out), error-handling also becomes very tedious in this callback-style system, as you need to add an error callback at each layer, and it's not easy to abort the rest of the processing in case of an error. What you'd ideally want is to list out the steps in linear function – which is what Promises are meant for.

Let's create a `pause` function, which takes in the number of seconds to pause for, and returns a Promise that is resolved after the appropriate number of seconds has elapsed. We will then be able to use it to fix the `setTimout` pyramid above.

To create a Promise, all you need to do is invoke the Promise constructor, passing in a function that represents the action you want taken. From somewhere within the action, you call `resolve(optionalSuccessValue)` or `reject(error)`. And so:

*A *pause* Promise*

```
function pause(seconds) {
    return new OfficeExtension.Promise(function(resolve, reject) {
        setTimeout(function() {
            resolve();
        }, seconds * 1000);
    });
}
```

By the way, in this particular case, the Promise has no way to fail – so we don't really need the `reject` callback. And moreover, because we're calling setTimout with an anonymous function whose sole job is to call another function, we can even simplify the `pause` function further, if we wanted to:

*A leaner *pause* Promise*

```
function pause(seconds) {
    return new OfficeExtension.Promise(function(resolve) {
        setTimeout(resolve, seconds * 1000);
    });
}
```

Armed with this newly-created function for creating Promises, we can now solve the rainbow scenario much more satisfactorily:

A Promise-ful rainbow

```
function createRainbow() {
    var $rainbow = $("#rainbow");

    $rainbow.css("background", "red");

    return pause(2)
        .then(function() {
            $rainbow.css("background", "orange");
            return pause(2);
        })
        .then(function() {
            $rainbow.css("background", "yellow");
            return pause(2);
        })
        ...
        .then(function() {
            $rainbow.css("background", "violet");
        });
}
```

By the way, you'll notice that there is an anomaly between how the first (red-background) call is made, relative to the rest. If we want to standardize our calls to all look the same, we could create an initial Promise, so that even the red-background call could be part of a `.then`. To create this starter Promise, simply call `Promise.resolve()`:

An more consistent Promise-ful rainbow, with a 'Promise.resolve()' at the start

```
function createRainbow() {
    var $rainbow = $("#rainbow");

    return Promise.resolve()
        .then(function() {
            $rainbow.css("background", "red");
            return pause(2);
        })
        .then(function() {
            $rainbow.css("background", "orange");
            return pause(2);
        })
        ...
}
```

With TypeScript 2.1's **async/await** feature – discussed in more detail in the the very next section – you can make these calls even cleaner:

*An Promise-ful rainbow with TypeScript's **async/await***

```
async function createRainbow() {
    var $rainbow = $("#rainbow");

    $rainbow.css("background", "red");
    await pause(2);

    $rainbow.css("background", "orange");
    await pause(2);

    ...

    $rainbow.css("background", "indigo");
    await pause(2);

    $rainbow.css("background", "violet");
}
```

Let’s get back to the *pause* function definition. As we saw, in order to wrap a callback-style function with a Promise, you create a new Promise, and put the entirety of the former callback-style code within the Promises’s constructor. Then, somewhere within the callback code (now residing in the constructor), whenever the asynchronous operation completes, you would call the `resolve` or `reject` functions. For `resolve`, you can pass in an optional value, which the caller of the Promise will be able to read when `await`-in the Promise (TypeScript), or inside of the `.then` function (JavaScript). [Similarly, you can pass an ‘Error‘ object to the ‘reject‘ function, which can be accessed by the caller as part of a ‘.catch‘ or a ‘try/catch‘ block]. So for example, if we wanted a Promise that, after a pause, returns a random number between 0 and 99, we could do:

A patience-strengthening random-number generator

```
function getRandomNumberAfterPause(seconds) {
    return new OfficeExtension.Promise(function(resolve, reject) {
        setTimeout(function() {
            resolve(Math.floor(Math.random() * 100));
        }, seconds * 1000);
    });
}
```

The TypeScript usage of such Promise would be as follows:

Obtaining the Promise’s resolved value using TypeScript

```
    let num = await getRandomNumberAfterPause(5 /*seconds*/)
    console.log(`After much anticipation, the lucky number is ${num}.`);
```

Using it with using conventional JavaScript would be as follows:

Obtaining the Promise's resolved value inside of a '.then' function

```
    getRandomNumberAfterPause(5 /*seconds*/)
        .then(function(result) {
            console.log("After much anticipation, " +
                "the lucky number is " + result + ".");
        });
```

Let's do one final example of a Promise-creation – this time, for a much more "real-world" use-case, of wrapping a jQuery AJAX call into a Promise. The concrete scenario is: given a list of stock name inputs (`["MSFT", "INTC", ...]`), call a web service, and return a dictionary of the latest stock prices (`{ "MSFT": 59.00, "INTC": 34.94, ... }`). Because jQuery use `.fail` instead of `.catch`, I find it best to wrap the AJAX call in a 100% official Promise – which we do here using the `resolve` and `reject` function.

Creating a new Promise for fetching stock-data, which ultimately wraps a jQuery AJAX call

```
function getStockData(stockNamesList) {
    let quotedCommaSeparatedNames = stockNamesList
        .map(name => `"${name}"`)
        .join(",");

    let url = '//query.yahooapis.com/v1/public/yql';
    let data = 'q=' +
        encodeURIComponent(
            'select * from yahoo.finance.quotes ' +
            'where symbol in (' + quotedCommaSeparatedNames + ')'
        ) +
        "&env=store%3A%2F%2Fdatatables.org" +
        "%2Falltableswithkeys&format=json";

    return new OfficeExtension.Promise(function(resolve, reject) {
        $.ajax({
            url: url,
            dataType: 'json',
            data: data,
            timeout: 5000
        })
        .done(function(result) {
            console.log("Web request succeeded");
            var stockDataDictionary = {};
            var stockDataArray = result.query.results.quote;
            stockDataArray.forEach(data => {
                var name = data['Symbol']
                var price = data['LastTradePriceOnly'];
                stockDataDictionary[name] = price;
            });

            resolve(stockDataDictionary);
        })
        .fail(function(error) {
            console.log("Web request failed:");
            console.log(url);

            reject(new Error(error.statusText));
        });
    });
}
```

I should note that it's possible to use a library like Q (https://github.com/kriskowal/q) to wrap the jQuery AJAX call. Personally, though, I've found it easier and more flexible to do my own wrapping, especially when I want to do some post-processing on the server response, like I did above.

4.3.3 Promises, `try/catch`, and `async/await`

When you read about using Promises in JavaScript, you will generally see examples that look as follows:

*A typical *JavaScript* Promise chain: asynchronous operation, a ‘.then‘, and a ‘.catch‘*

```
// Attach a button-click to a function (using jQuery):
$("#retrieve-data").click(retrieveAndProcessData);

function retrieveAndProcessData() {
    $("#progress-overlay").show();

    fetchRemoteData()
        .then(function(data) {
            // ...
            // some analysis or processing of the data
            // ...

            // Note that you must either ensure that this
            // whole processing is synchronous in nature,
            // or you must return another Promise.
        })
        .catch(function(e) {
            $("#error").text("Error: " + e).show();
        })
        .then(function() {
            $("#progress-overlay").hide();
        });
}
```

A few things to note:

1. You start with some asynchronous operation (e.g., waiting on a network call)
2. In order to “chain” that asynchronous operation with the logic that follows (and that await the completion of the preceding task), you create a `.then` statement, and write your code within that. The function takes

in a parameter (`data` in the example above), which is the value passed in from the previous function.

3. The `.then` either executes synchronously (e.g., display data and be done), or if it needs to do another asynchronous action, it would need to return a Promise (e.g., `return updateMissingValues(data)`). Failing to do so will "break the Promise chain" (setting the new async operation loose) – with many ill practical effects, such as swallowed errors and out-of-order execution (e.g., the progress overlay lifting before the operation is done).
4. Promises can be chained indefinitely. I.e., there could be another `.then` waiting beneath the current one, esp. if the former returned a Promise.
5. Each `.then` *can* be followed by a `.catch`, which would handle errors from *this* `.then` or anything further upstream. This means that you can localize error-handling to only a particular portion of the code, if you wish. In most cases I've seen, though, there is simply a single `.catch` in the end (which means it will catch an error that occured *anywhere* up the chain), and then handle the error there (often by displaying the error, or some massaged version thereof, to the user). It's fine to have just the single `.catch` – but it is vitally important to have it, as silent errors are the bain of a programmer's / JavaScripter's existence.

The above was pure JavaScript. **Now, enter TypeScript 2.1, and the async/await keywords**. The usage is very simple: **anytime there was a Promise you wanted to `.then` on, you now simply `await` it**. Furthermore, in any function where you have an `await`, you **mark the function with the keyword `async`** (which, by the way, internally turns the function into a Promise).

Now, let's see the TypeScript version:

*A **async/await**-ified Promise chain*

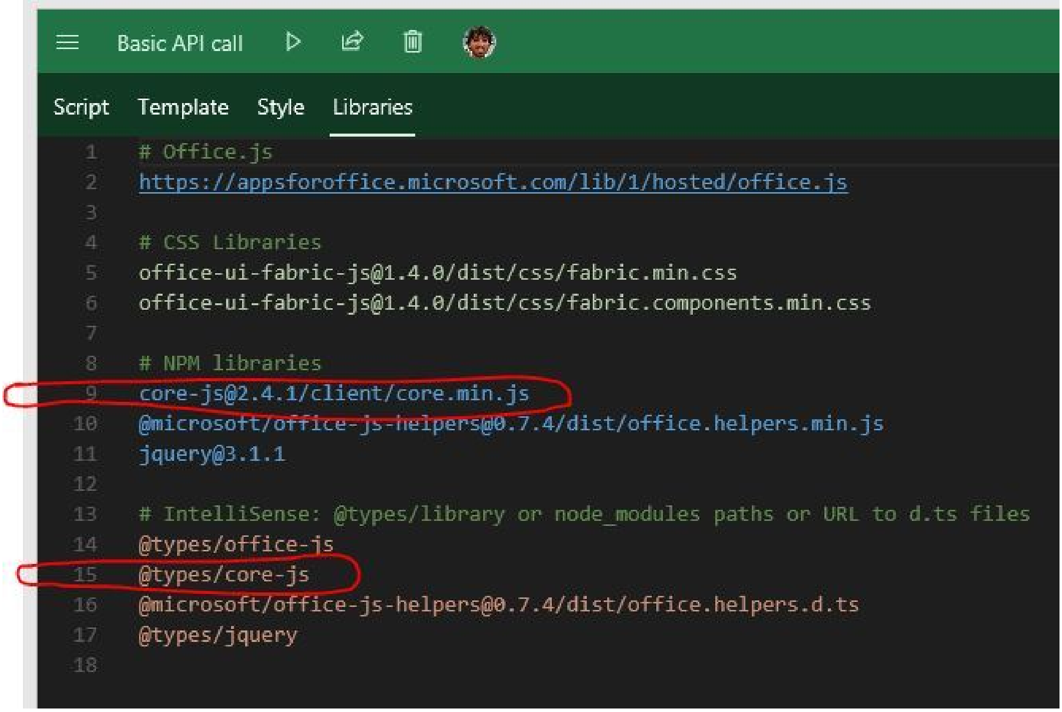
```
// Attach a button-click to a function (using jQuery):
$("#retrieve-data").click(retrieveAndProcessData);

async function retrieveAndProcessData() {
    $("#progress-overlay").show();

    try {
        let data = await fetchRemoteData();

        // ...
        // Do your same analysis or processing of the data
        // ...

        // If you require to do another
        // asynchronous operation, await it as well.

    } catch (e) {
        $("#error").text("Error: " + e).show();
    }

    $("#progress-overlay").hide();
}
```

A few things to note about this simplified TypeScript rendition:

1. Awaiting for a process to complete before moving on is accomplished via a single word, **await** (`await fetchRemoteData()`). The return value, which previously went as a parameter to the function beneath, is now just being assigned-to directly.
2. You can await anything that is a Promise (or that is another function marked with the word `async`, since those ultimately get turned into Promises). The interop between is Promises and async/await is quite beautiful.
3. Instead of needing to do a `.catch`, you can use regular try/catch blocks, and let the compiler do its magic. The same goes for having a statement (e.g., `$("#progress-overlay").hide()`) following the try/catch block: you can simply put it as if it's normal sequential logic.
4. One important bit of fine-print: you can only use `await` inside functions marked as `async` (and those functions, in turn, must be in places where async functions – or more precisely, Promises – are expected). Thus, for example, you *cannot* pass an `async` function to an `array.forEach`, since that one expects normal functions, not Promises.

Programming with `async/await` feels beautifully natural: again, at the risk of sounding like a broken-record, I highly encourage you to give TypeScript a try, especially in light of the asynchronous nature of the Office.js APIs.

One gotcha: the `async` keyword is effectively creating a Promise for you, which means that you need to have the global `window.Promise` object defined. In Script Lab, we include the library core-js by default (option #3), which provides a Promise polyfill:



```
# Office.js
https://appsforoffice.microsoft.com/lib/1/hosted/office.js

# CSS Libraries
office-ui-fabric-js@1.4.0/dist/css/fabric.min.css
office-ui-fabric-js@1.4.0/dist/css/fabric.components.min.css

# NPM libraries
core-js@2.4.1/client/core.min.js
@microsoft/office-js-helpers@0.7.4/dist/office.helpers.min.js
jquery@3.1.1

# IntelliSense: @types/library or node_modules paths or URL to d.ts files
@types/office-js
@types/core-js
@microsoft/office-js-helpers@0.7.4/dist/office.helpers.d.ts
@types/jquery

```

If you prefer not to bring in an extra library, you can also rely on the fact that Office.js (as long as you're using ExcelApi 1.2+ or WordApi 1.2+) includes a Promise library. So all you need to do, if you want, is add the following one-liner somewhere inside of `Office.initialized`.

Using the Office-provided Promise library for 'async/await'

```
Office.initialized = () => {
    // ... possibly other initialization logic

    (window as any).Promise = OfficeExtension.Promise;
}
```

Note that if a Promise is already globally-defined, the `OfficeExtension.Promise`

will use it instead of its own version, so at that point
`(window as any).Promise = OfficeExtension.Promise`
effectively become a no-op equivalent to
`(window as any).Promise = (window as any).Promise`.
But I can't think of any reason why that would be harmful, either. So if you want to, go ahead and use the Office-provided Promise, or include a polyfill of your choice.

JavaScript-only

For folks using JavaScript instead of TypeScript, you may find it helpful to learn of another technique for Promises: "***Passing in functions to Promise*** **`.then`*-functions***"

5. Office.js APIs: Core concepts

Chapter structure

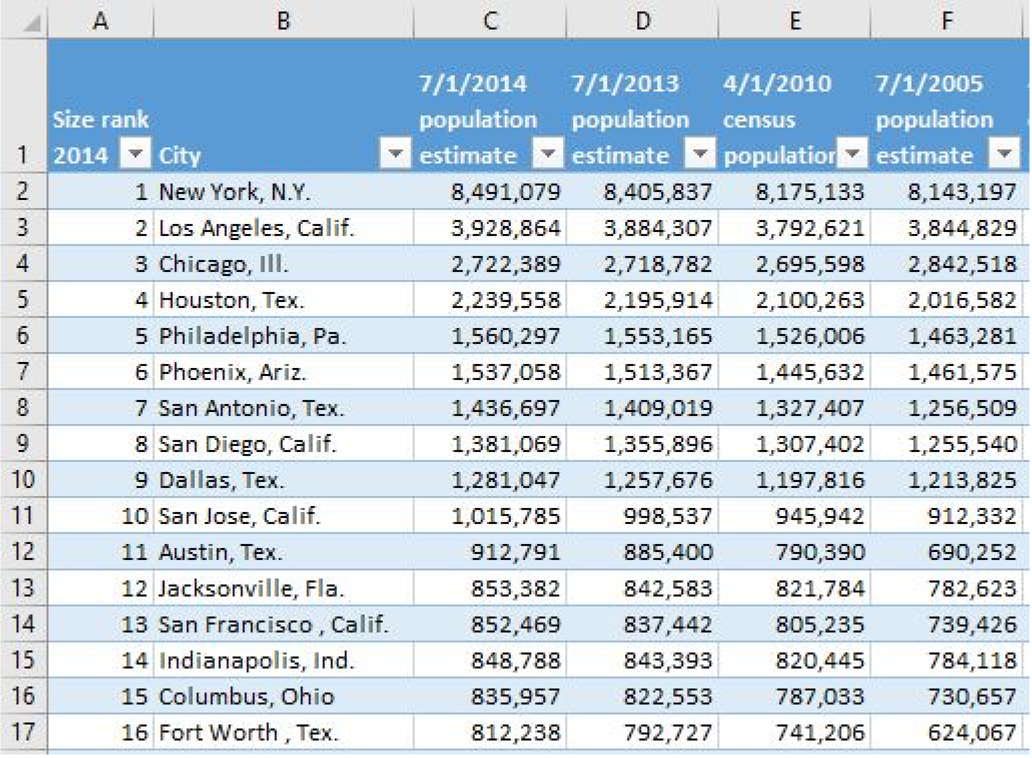
- Canonical code sample: reading data and performing actions on the document
- Excel.run (Word.run, etc.)
- Proxy objects: the building-blocks of the Office 2016 API model
  - Setting document properties using proxy objects
  - The processing on the *JavaScript side*
  - The processing on the *host application's side*
  - Loading properties: the bare basics
  - De-mystifying `context.sync()`
- Handling errors
- Recap: the four basic principles of Office.js

5.1 Canonical code sample: reading data and performing actions on the document

To set the context for the sort of code you'll be writing, let's take a very simple but canonical example of an automation task. This particular example will use Excel, but the exact same concepts apply to any of the other applications (Word, OneNote) that have adopted the new host-specific/Office 2016+ API model.

Scenario: Imagine I have some data on the population of the top cities in the United States, taken from "Top 50 Cities in the U.S. by Population and Rank" at http://www.infoplease.com/ipa/a0763098.html. The data – headers and all, just like I found it on the website – describes the population over the last 20+ years.

Let's say the data is imported into Excel, into a table called "PopulationData". The table could just as easily have been a named range, or even just a selection – but having it be a table makes it possible to address columns by name rather than index. Tables are also very handy for end-users, as they can filter and sort them very easily. Here is a screenshot of a portion of the table:



	A	B	C	D	E	F
1	Size rank 2014	City	7/1/2014 population estimate	7/1/2013 population estimate	4/1/2010 census populatior	7/1/2005 population estimate
2	1	New York, N.Y.	8,491,079	8,405,837	8,175,133	8,143,197
3	2	Los Angeles, Calif.	3,928,864	3,884,307	3,792,621	3,844,829
4	3	Chicago, Ill.	2,722,389	2,718,782	2,695,598	2,842,518
5	4	Houston, Tex.	2,239,558	2,195,914	2,100,263	2,016,582
6	5	Philadelphia, Pa.	1,560,297	1,553,165	1,526,006	1,463,281
7	6	Phoenix, Ariz.	1,537,058	1,513,367	1,445,632	1,461,575
8	7	San Antonio, Tex.	1,436,697	1,409,019	1,327,407	1,256,509
9	8	San Diego, Calif.	1,381,069	1,355,896	1,307,402	1,255,540
10	9	Dallas, Tex.	1,281,047	1,257,676	1,197,816	1,213,825
11	10	San Jose, Calif.	1,015,785	998,537	945,942	912,332
12	11	Austin, Tex.	912,791	885,400	790,390	690,252
13	12	Jacksonville, Fla.	853,382	842,583	821,784	782,623
14	13	San Francisco , Calif.	852,469	837,442	805,235	739,426
15	14	Indianapolis, Ind.	848,788	843,393	820,445	784,118
16	15	Columbus, Ohio	835,957	822,553	787,033	730,657
17	16	Fort Worth , Tex.	812,238	792,727	741,206	624,067

The population data, imported into an Excel table

Now, suppose my task is to find the top 10 cities that have experienced the most growth (in absolute numbers) since 1990. How would I do that?

The code in the next few pages shows how to perform this classic automation scenario. As you look through the code, if not everything will be immediately obvious – and it probably won't be – don't worry: the details of this code is what the rest of this chapter (and, to some extent, the book) is all about! But I think it's still worth reading through the sample as is for the first time, to gain a general sense of how such task would be done via Office.js.

Note: In a more real-world scenario, this sample would be broken down into three or more functions (one to read the data and calculate the top 10 changed cities; another to write out the table and chart; and a third main function to orchestrate the other two). For purposes of this sample, though – and in order to make it easily readable from top to bottom, rather than having the reader jump back and forth – I will do it in one long function. If you try out the sample code in Script Lab, I have both snippet varieties: one identical to the code below, and another with refactored code. In a later section, "*A more complex* `context.sync` *example*", I will show another example of code where I do split out the tasks into smaller subroutines.

TIP: For those coming from VBA, one thing you'll immediately see – and what you'll need to adjust to – is that ***everything is zero-indexed***. For example, `worksheet.getCell(0, 0)`, which would be absolutely incorrect and throw an error in VBA, is the correct way to retrieve the first cell using the Office.js APIs.

Of course, for user-displayable things like the address of the cell, it would still report "`Sheet1!A1`", since that's what the user would have seen. But in terms of programmatic access, *everything* is ***zero-indexed***[1].

[1] There is hardly a rule without an exception. In Excel in particular, the API exposes the native Excel function to JavaScript under the `workbook.functions` namespace (i.e., `workbook.functions.vlookup(...)`. Within such functions, it made sense to keep the indexing consistent with the sort of native Excel formulas that a user would type (otherwise, what's the point?!) – and so there, any index is **1-indexed**. For example, if the third parameter to a "vlookup" call, "colIndexNum", equals to 3, this refers to the third column, *not* the 4th one. But in terms of the object model, everywhere else (and in JavaScript itself, of course!), everything is zero-indexed.

A TypeScript-based, canonical data-retrieval-and-reporting automation task

```
const RawDataTableName = "PopulationTable";
const OutputSheetName = "Top 10 Growing Cities";

await Excel.run(async context => {
    // Here and elsewhere: Create proxy objects to represent
    // the "real" workbook objects that we'll be working with.
    let originalTable = context.workbook.tables
        .getItem(RawDataTableName);

    let nameColumn = originalTable.columns.getItem("City");
    let latestDataColumn = originalTable.columns.getItem(
        "7/1/2014 population estimate");
    let earliestDataColumn = originalTable.columns.getItem(
        "4/1/1990 census population");

    // Now, load the values for each of the three columns that we
    // want to read from.  Note that, to support batching operations
    // together, the load doesn't *actually* happen until
    // we do a "context.sync()", as below.

    nameColumn.load("values");
    latestDataColumn.load("values");
    earliestDataColumn.load("values");

    await context.sync();


    // Create an in-memory data representation, using an
    // array with JSON objects representing each city.
    let citiesData: Array<{ name: string, growth: number }> = [];

    // Start at i = 1 (that is, 2nd row of the table --
    // remember the 0-indexing) in order to skip the header.
    for (let i = 1; i < nameColumn.values.length; i++) {
        let name = nameColumn.values[i][0];

        // Note that because "values" is a 2D array
        // (even if, in this case, it's just a single
        //  column), extract the 0th element of each row.
        let pop1990 = earliestDataColumn.values[i][0];
        let popLatest = latestDataColumn.values[i][0];

        // A couple of the cities don't have data for 1990,
        // so skip over those.
        if (isNaN(pop1990) || isNaN(popLatest)) {
            console.log('Skipping "' + name + '"');
        }

        let growth = popLatest - pop1990;
        citiesData.push({ name: name, growth: growth });
    }

    let sorted = citiesData.sort((city1, city2) => {
        return city2.growth - city1.growth;
        // Note the opposite order from the usual
        // "first minus second" -- because want to sort in
        // descending order rather than ascending.
    });
    let top10 = sorted.slice(0, 10);

    // Now that we've computed the data, create
    // a report in a new worksheet:
    let outputSheet = context.workbook.worksheets
        .add(OutputSheetName);
    let sheetHeaderTitle = "Population Growth 1990 - 2014";
    let tableCategories = ["Rank", "City", "Population Growth"];

    let reportStartCell = outputSheet.getRange("B2");
    reportStartCell.values = [[sheetHeaderTitle]];
    reportStartCell.format.font.bold = true;
    reportStartCell.format.font.size = 14;
    reportStartCell.getResizedRange
        (0, tableCategories.length - 1).merge();

    let tableHeader = reportStartCell.getOffsetRange(2, 0)
        .getResizedRange(0, tableCategories.length - 1);
    tableHeader.values = [tableCategories];
    let table = outputSheet.tables.add(
        tableHeader, true /*hasHeaders*/);

    for (let i = 0; i < top10.length; i++) {
        let cityData = top10[i];
        table.rows.add(
            null /* null means "add to end" */,
            [[i + 1, cityData.name, cityData.growth]]);

        // Note: even though adding just a single row,
        // the API still expects a 2D array (for
        // consistency and with Range.values)
    }

    // Auto-fit the column widths, and set uniform
    // thousands-separator number-formatting on the
    // "Population" column of the table.
    table.getRange().getEntireColumn().format.autofitColumns();
    table.getDataBodyRange().getLastColumn()
        .numberFormat = [["#,##"]];


    // Finally, with the table in place, add a chart:
    let fullTableRange = table.getRange();

    // For the chart, no need to show the "Rank", so only use the
    //     column with the city's name, and expand it one column
    //     to the right to include the population data as well.
    let dataRangeForChart = fullTableRange
        .getColumn(1).getResizedRange(0, 1);

    let chart = outputSheet.charts.add(
        Excel.ChartType.columnClustered,
        dataRangeForChart,
        Excel.ChartSeriesBy.columns);

    chart.title.text = "Population Growth between 1990 and 2014";

    // Position the chart to start below the table, occupy
    // the full table width, and be 15 rows tall
    let chartPositionStart = fullTableRange
        .getLastRow().getOffsetRange(2, 0);
    chart.setPosition(chartPositionStart,
        chartPositionStart.getOffsetRange(14, 0));

    outputSheet.activate();

    await context.sync();

}).catch((error) => {
    console.log(error);
    // Log additional information, if applicable:
    if (error instanceof OfficeExtension.Error) {
        console.log(error.debugInfo);
    }
});
```

Try it out

If you want to follow along with the code above, just import one of the following snippet IDs into Script Lab.
1. Same code as above (except a small tweak to the error-handling):
aa81d73587f62f35e46ad6a904bb20df
2. A refactored version, which breaks apart the functions into multiple subroutines:
fb8913d0a899c88e3ea82773a135dfd0

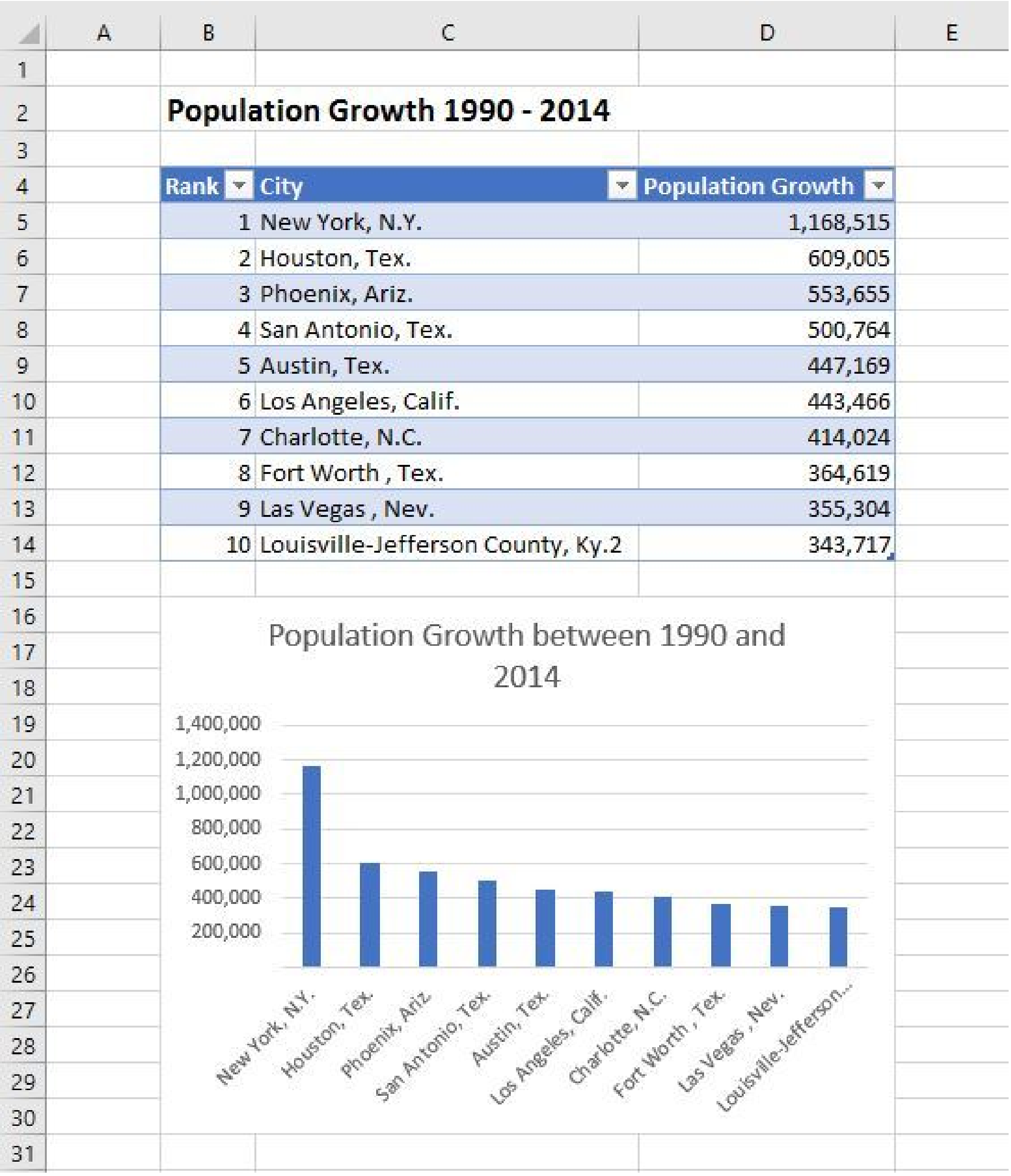
The Script Lab snippets contain a “setup” button which will populate the workbook with the necessary data. But if you prefer to download it as a file, you can find it here: **http://www.buildingofficeaddins.com/samples/population-analysis**

JavaScript-only

For folks using JavaScript instead of TypeScript, you will find it useful to reference the ***JavaScript-only version of the canonical code sample***.

For those coming from VBA or VSTO, by far and away the biggest difference you'll notice is the need to explicitly call out which properties you want loaded (`nameColumn.load("values")`), and the two `await context.sync()`-s towards the beginning and very end of the operations. But for the rest of the logic, you have simple sequential code, not unlike VBA. This is the beauty of the new 2016+ APIs – that they provide you with local proxy objects that "stand in" for document objects, and allow you to write mostly-synchronous code (interspersed with the occasional "load" and "sync"). You will read more about loading and syncing in the forthcoming sections.

If you run the above code, here is what the resulting sheet looks like:



Population Growth 1990 - 2014

Rank	City	Population Growth
1	New York, N.Y.	1,168,515
2	Houston, Tex.	609,005
3	Phoenix, Ariz.	553,655
4	San Antonio, Tex.	500,764
5	Austin, Tex.	447,169
6	Los Angeles, Calif.	443,466
7	Charlotte, N.C.	414,024
8	Fort Worth , Tex.	364,619
9	Las Vegas , Nev.	355,304
10	Louisville-Jefferson County, Ky.2	343,717

Now, let’s dive in and see how this sample works.

5.2 Excel.run (Word.run, etc.)

Let's start with the very first line of the code: `Excel.run`. Because the sample code is based in an Excel scenario, I will use the term `Excel.run` for the remainder of this chapter – but the concept is 100% equivalent to `Word.run`, `OneNote.run`, etc.)

`Excel.run` is a function that, in turn, accepts a function as its input parameter. This inner function – typically referred to as the *batch function* – describes the work that we'd like Excel to do. When you look at the population-data sample from the previous section, almost the entire code – other than the `Excel.run` line at the beginning, and the `.catch(...)` block at the very end – is part of this *batch function*.

The batch function accepts a *request context* parameter, which is the means by which the communication with the host happens. In the population-data sample, you saw the context used in two ways:

1. Use the `context` to fetch the `workbook` object (or `document` in Word, etc.), and do the other object-navigation from there. A couple examples:

   ```
   var table = context.workbook.tables.getItem("PopulationTable");
   ```

   ```
   var outputSheet = context.workbook.worksheets.add(
    "Top 10 Growing Cities");
   ```

2. Use the `context` as part of a `context.sync()` – which will be covered shortly in a subsequent section, but, at a high-level, is essential in order to "commit" any local changes back into the document.

You can think of the `Excel.run` as a boiler-plate preamble; **it is the batch function that is the star of the show**, but it needs a *request context* in order to start off the chain of OM calls. The `Excel.run` call is simply a means to that end, as it creates a new *request context* that it *passes into* the batch function. Because nothing can happen without a *request context*, you will see that essentially all code in this book and in the documentation begin with this

`Excel.run` (or `Word.run`, etc.) incantation[2].

Thus, the typical and simplest use-case for the `run` method for TypeScript-based code is:

A simple 'Excel.run', with a '.catch' at the end

```
    Excel.run(async (context) => {
        // ...
        await context.sync();

    }).catch(...);
```

In the code above, you can see `async/await` being used *within* the `Excel.run`, but the call to `run` itself is done using a regular Promise pattern (namely, without an `await` in front of it, and with a `.catch` *function* at the end). If you prefer, you can instead wrap the whole thing in a `try/catch` block, making sure to await the completion of `Excel.run`, as follows:

The same 'Excel.run', but this time 'await'-ed and wrapped in a 'try/catch' block

```
    try {
        await Excel.run(async function(context) {
            // ...
            await context.sync();
        });
    } catch (error) {
        // ...
    }
```

It doesn't matter which approach you choose: the only important thing is to ensure that you don't let the asynchronicity spin out of control – so you

[2] There is a very particular use-case where, for performance reasons, you may need to avoid `Excel.run` and use a *raw* request context instead. In the entire time that I've been answering StackOverflow questions, I've seen exactly *one* real-world example that would have benefited from this. I will describe this use-case later in the book, in "*When to use a raw RequestContext in place of an Excel.run*", but I don't think you should worry about this too much for now. If you *are* in fact experiencing performance issues, there are a number of other more likely performance offenders, which I describe later on in the book, in "*Summary: writing efficient code*".

must either `await` the `Excel.run`, or use `.then`-s and `.catch`-es on it. The part about catching errors is particularly important, as there are a wide array of circumstances under which API calls will fail (especially while you're still developing the application). Thus, while I'm a big fan of `async/await` and 100% recommend using it *inside* of `Excel.run`, my worry with using it outside is that it's *really easy* to forget the `await` in front of `Excel.run`. If you do, you will be left mistakenly confident that you're protected by the `catch` statement, whereas in actuality the `catch` does nothing if you forgot to `await` the asynchronous portion[3].

Whichever way you go, for purposes of getting started[4], that is really all that you need to know about `Excel.run`. So, with this `Excel.run` preamble out of the way, the subsequent sections will focus on what you can do *within* the *batch function*, after the process has been jump-started by the `Excel.run`.

[3] I actually had this happen to me right as I was re-writing the "canonical sample" of this chapter, TypeScript-ifying my previously-JavaScript-based content. It was clear that the API was stopping midway due to some issue, and yet there was no indication of the error! Once I noticed the omission, I promptly switched my still-being-TypeScript-ified guidance to recommend `async/await` inside the `run`, but not outside of it.

[4] The `Excel.run` function also has a number of other overloads, which are covered in detail in "`Excel.run` *(*`Word.run`*, etc.) advanced topics*".

5.3 Proxy objects: the building-blocks of the Office 2016 API model

5.3.1 Setting document properties using proxy objects

At the fundamental level, the Office 2016+ API model consists of local JavaScript *proxy* objects, which are the local placeholders for real objects in the document. You get a proxy object by calling some property or method (or a chain of properties or methods) that originate somewhere off of the *request context*. For example, if you have an Excel object called `workbook`, you can get cells `A1:B2` on `Sheet1` by calling

```
var myRange = workbook.worksheets.getItem("Sheet1")
    .getRange("A1:B2");
```

To apply values to the object, you would simply set the corresponding properties. For example:

```
myRange.values = [["A", "B"], [1, 2]];
myRange.format.fill.color = "yellow";
```

The same applies to calling a method – you simply invoke it. For example,

```
myRange.clear();
```

Importantly, what happens beneath the covers is that the object – or, more accurately, the *request context* from which the object derives – accumulates the changes locally, much like a changelist in a version-control system. **Nothing is "committed" back into the document until you recite the magical incantation:**

```
context.sync();
```

5.3.2 The processing on the *JavaScript side*

Let's pause for a moment. If all of these JavaScript objects are simply proxy objects, and all you're doing is queuing up commands, how long can you keep going? It turns out that as long as you don't need any information back from the document (i.e., you're not reading back some values and then multiplying or formatting or doing whatever else with them), you can create an arbitrarily-long queue of commands. In the population-data example, after reading back the initial batch of data, I issued a bunch of commands all at once: creating a new sheet, setting some values on it, overlaying it with a table and adding a bunch of rows, formatting the table, adding and positioning a chart, and finally setting focus on the newly-created output sheet.

On the JavaScript side, whenever I called a *property* that involved primitive types – so, for example, setting `sheetHeader.values` – the action would get noted and added to the internal queue, and then the JavaScript operations would proceed as is. Likewise, whenever I called a `void` *returning method* – such as `sheetHeader.merge()` – this too would get internally noted and added to the queue. If I called a *property or method* ***that returns an API object***, though – for example, `context.workbook.worksheets.add(...)`, or `outputSheet.getRange("B2:D2")` – Office.js would go and **create a new on-the-fly proxy object** and return it to me, noting internally how this object was derived.

In all those cases, the overarching paradigm is that Office.js tries to *create the illusion* that the operations are synchronous, even though in reality it simply added them to an internal queue, and waits until they can be "flushed" as part of a `context.sync()`.

5.3.3 The processing on the *host application's side*

The previous section described what happens on the JS layer for outgoing commands. But what happens on the *receiving* end? On the host application's side (a.k.a, "the server"), the long chain of commands is executed one-by-one, resolving any proxy objects as the need arises (*"supply on demand"?*). So for example, when a call comes in to add a sheet, the resulting object is internally stored for the duration of the batch; and then, when this object gets used in a subsequent line for retrieving a range, this range is likewise created (and internally identified as being created off of a particular parent object, using method such-and-such with parameters such-and-such). When a subsequent call comes in to set values of the range, this call – which in turn references the object that it's operating on – is identified to mean the range that was just recently created. And so forth.

FUN FACT: This ability to call methods or properties on not-yet-created objects is sometimes known as “time travel”, since you’re effectively getting the result of a method call (a new object) before the batch request is ever made! Since it has such a cool name, let’s do another example:

```
var newSheet =
    context.workbook.worksheets.add("NewSheet");

// NewSheet technically doesn't exist yet. But that doesn't
// prevent us from calling methods on it, because by the time
// that everything executes -- in order -- it *will* exist.

var dataRange = newSheet.getRange("A1:B2");
dataRange.values = [["A", "B"], [1, 2]];
var chart = newSheet.charts.add(
    Excel.ChartTypes.pie, dataRange);

// Again, the chart doesn't exist yet...
// But we can still keep going!

chart.title.text = "My Data";

// ...

context.sync().catch(...);
```

If this all seems magical – well, to some extent it down right is! But it is important to remember that until you call `context.sync()`, you’re effectively building castles in the sky. And, an interesting corollary to this is that even *erroneous* code (such as accessing a non-existing item) won’t throw until it’s part of the `context.sync()` execution – up to that point, the changes are merely queued up. So for example, you can easily write the following code:

Oh-oh. Code executing past an obvious OM error

```
context.workbook.worksheets.getItem
    ("SheetThatDoesNotExist").getRange("A1:B2").clear();
console.log("I'm past the call");
context.workbook.getSelectedRange().format.fill.color = "yellow";

context.sync().then(...).catch(...);
```

When you run the above code, execution at the JavaScript layer will happily keep going past the `SheetThatDoesNotExist` line to the next JavaScript call: `console.log("I'm past the call)"`. And in fact, it will keep on going all the way through till `context.sync()`.

This is probably not what you'd expect – after all, shouldn't the invalid-sheet-fetching throw an error? In a synchronous VBA/VSTO-like world, it absolutely would. But remember, in this proxy-object/batched/deferred-execution world of Office.js, all you're doing is building up an array of commands, but not yet executing them! As discussed in *"Why is Office.js async?"*, querying the document every time to perform a document-object-model request would be impractical. And so, without that direct feedback, the JavaScript layer has no way of knowing that `SheetThatDoesNotExist` does not exist in the workbook. Only when the JS layer is waiting on the asynchronous completion of `context.sync()` – and when the host application is meanwhile processing the queue of commands – does the error become apparent. At that point, the execution halts on the host applications' side, dropping any other calls that were queued up in the batch (in this case, setting the selected range to yellow). The host then signals back to the JavaScript engine to return a failure in the form of a rejected promise. But importantly, any non-API-object JavaScript calls that preceded `context.sync()` will have already run!

This is the trickiest part of the *new* Office.js's async model, and the most common pitfall (especially if you accidentally let a `context.sync()` run loose without awaiting it, thereby executing a bunch of JavaScript before the document object-model calls came back). So remember: **it's all proxy objects – and nothing happens until you call `context.sync()"`!**

5.3.4 Loading properties: the bare basics

The preceding section only talked about setting properties or calling methods on document objects. What about when you need to ***read data back*** from the document?

For reading back document data, there is a special command on each object, `object.load(properties)`. An identically-functioning method can also be found on the context object: `context.load(object, properties)`. The two are 100% equivalent (in fact, `object.load` calls `context.load` internally), so which one you use is purely stylistic preference. I generally prefer `object.load`, just because it feels like I'm dealing with the object directly, rather than going through some context middleman/overlord. But again, it's purely stylistic.

The load command is queued up just like any of the others, but at the completion of a `context.sync()` – which will be covered in greater detail in a later section – an internal post-processing step will automatically populate the object's properties with the requested data. The property names that you pass in into the `object.load(properties)` call are the very properties that – after a `sync` – you will be able to access directly off of the object.

For example, if you look at the IntelliSense for the Excel `Range` object, you will see that it contains properties like `range.values` and `range.numberFormat`. If you need these values and number formats in order to execute your function (maybe to display them to the user, or to calculate the minimum/maximum, or to copy them to a different section of the workbook, or to use them to make a web service call, etc.), you would simply list them as a comma-separated list in the `.load` function, as show in the example below:

Loading and using object properties

```
Excel.run(async (context) => {
    let myRange = context.workbook.worksheets
        .getItem("Sheet1").getRange("A1:B2");

    myRange.load("values, numberFormat");

    await context.sync();

    console.log(JSON.stringify(myRange.values));
    console.log(JSON.stringify(myRange.numberFormat));

}).catch(...);
```

TIP: In the above code, you see the use of `JSON.stringify`. For those relatively new to JavaScript: `JSON.stringify`, used above, is a very convenient way of viewing an object that might otherwise show up as `[object Object]` (typically a complex type or a class). The `JSON.stringify` also helps for types that might otherwise show up incorrectly, like a nested array (for example, the array `[[1,2],[3,4]]` shows up as the string `1,2,3,4` if you call `.toString()` on it, instead of its true representation of `[[1,2],[3,4]]`).

Importantly, just as in the code above, the reason to call `object.load` is that you ***intend to read back the values*** in the code following the `context.sync()`, and that your operation can't continue without having that information. This brings us to arguably the most important rule in all of Office.js:

THE GOLDEN RULE OF `OBJECT.LOAD(PROPERTIES)`: You ***ONLY*** need to load the object ***IF you are intending to read back its PRIMITIVE PROPERTIES (numbers, strings, booleans, etc.)***.

Just calling methods on the object, or setting its properties, or using it to navigate to another object, or doing ***anything else that doesn't involve reading and using the object's property values*** does ****NOT**** require a `load`.

By the way, the **return value of `object.load(...)`** is the object itself. This is done purely for convenience, saving a line or two of vertical space. That is,

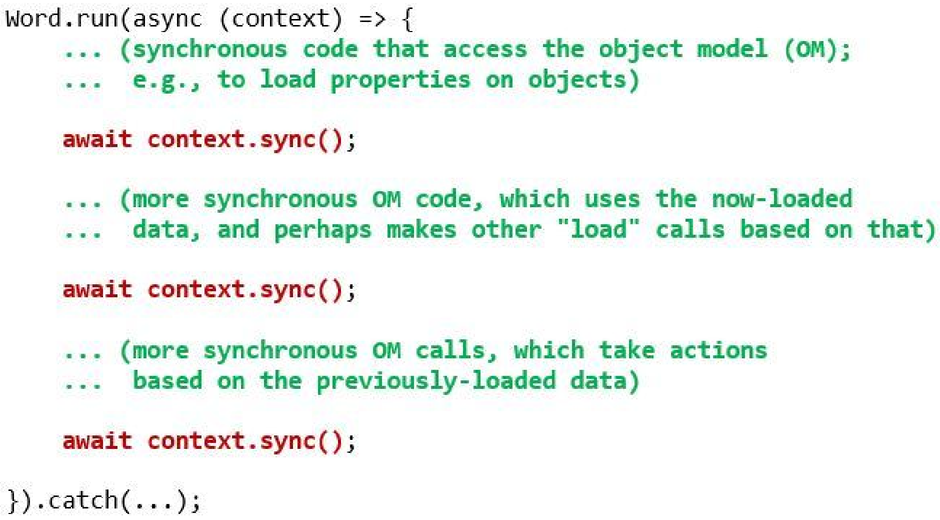
```
let range = worksheet.getRange("A1").load("values");
```

Is 100% equivalent to the more verbose version:

```
let range = worksheet.getRange("A1");
range.load("values");
```

5.3.5 De-mystifying `context.sync()`

Throughout the preceding sections, you will have seen the term – and seen example usage of – `context.sync()`. The purpose of `context.sync()` is for the developer to declare that he/she is done with either the entire batch or a portion of it. An invocation of `context.sync()` returns a Promise object, which can then be `await`-ed or `.then`-ed with follow-up operations (which in turn might also call to `context.sync()` to signal the completion of *their* portion of the batch). Importantly, chaining Promises together returns a *meta*-Promise, which *resolves* only when all of the child Promises have finished chaining. This Promise is what must be returned out of the batch function, so that `.run`-method knows that the OM operations have completed[5].



```
Word.run(async (context) => {
    ... (synchronous code that access the object model (OM);
    ...  e.g., to load properties on objects)

    await context.sync();

    ... (more synchronous OM code, which uses the now-loaded
    ...  data, and perhaps makes other "load" calls based on that)

    await context.sync();

    ... (more synchronous OM calls, which take actions
    ...  based on the previously-loaded data)

    await context.sync();

}).catch(...);
```

As an add-in author, **your job is to minimize the number of `context.sync()` calls**. Each `sync` is an extra round-trip to the host application; and when that application is Office Online, the cost of each of those round-trip adds up quickly.

If you set out to write your add-in with this in principle in mind, you will find that you need *a surprisingly small number of* `sync` *calls*. In fact, when writing this chapter, I found that I really needed to rack my brain to come

[5] See "*What does* `.run` *do, and why do I need it?*" regarding what `.run` does with this information.

up with a scenario that *did* need more than two `sync` calls. The trick for minimizing `sync` calls is to arrange the application logic in such a way that you're initially scraping the document for whatever information you need (and queuing it all up for loading), and then following up with a bunch of operations that modify the document (based on the previously-loaded data). You've seen several examples of this already: one in the intro chapter, when describing why Office.js is async; and more recently in the "canonical sample" section at the beginning of this chapter. For the latter, note that the scenario itself was reasonably complex: reading document data, processing it to determine which city has experienced the highest growth, and then creating a formatted table and chart out of that data. However, given the "time-travel" superpowers of proxy objects, you can still accomplish this task as one group of *read* operations, followed by a group of *write* operations.

Still, there are some scenarios where multiple loads may be required. And in fact, there may be legitimate scenarios where even doing an extra `sync` is the right thing to do – if it saves on loading a bunch of unneeded data. You will see an example of this later in the chapter.

Looking for a JavaScript-based (non-`async`/`await`) template instead, for a multi-`sync` scenario?

For folks using plain ES5 JavaScript instead of TypeScript, see an analogous image in one of the Appendix sections: "*A JavaScript-based multi-sync template*".

5.4 Handling errors

5.4.1 The basics of Office.js errors

If and when an error does occur, the most important thing is to *be aware that this has happened*. Silent errors, esp. in developing with Office.js, can easily happen if you forget to add appropriate error handling.

In the example in at the very start of this chapter (see "*Canonical code sample*") – and really, anywhere else throughout the book and online samples – one thing to note is the presence of a `.catch` statement. The `.catch` traps any errors that could have occurred anywhere within the ".run" operation. As explained in "*Promises, try/catch, and async/await*", a `.catch` is the Promise equivalent of a `try/catch` block. With TypeScript, you can use either, but I personally find it easier and more concise to use `.catch`-es outside of the `Excel.run` (again, see the referenced section for an explanation of my rationale for this).

When writing quick prototype code, it's easy to fall into the trap of skipping error handling. **Don't**. It's literally less than 50 extra characters.

```
.catch(function(e) {
    console.log(e);
});
```

In fact, in its most basic form[6], you can make it just 20 extra characters:

```
.catch(console.log);
```

Trust me, I've seen it over and over again: the five extra seconds it takes to type a `catch` function/statement, relative to the time spent on head-scratching

[6] Why does this work? Remember that functions are first-class citizens in JavaScript. The `.catch` (or `.then`, for that matter) expects to be given a function that it can call, passing in the error object as a parameter. You can define the function as an anonymous lambda function, as in the previous example, but if all you're doing is turning around and calling another function, there's hardly a point to wrapping the call. You can instead just pass in the function itself to the `.catch`.

and pondering why nothing seems to be happening, is well worth the investment.

If the error comes from the Office.js APIs (as opposed to a JavaScript runtime error, or a failed ajax request, etc.), it will contain an **error code**, **message**, and often a useful **.debugInfo** property. Let me describe each of them in detail:

- The **`code`** property is a language-invariant (i.e., always-English) error code, such as “`InvalidArgument`”. Error codes can be found and compared against the list in `OfficeExtension.ErrorCodes`, as well as the host-specific error codes (`Excel.ErrorCodes`, `Word.ErrorCodes`, etc.)

- The **`message`** property is a localized string that describes the error in a little more detail, but is still *human-friendly*. If you let the message fall through to the end-user[7], the text won’t scare them (though it might not be helpful to them, either).

- The default **`toString`** representation of the error will give you “<ErrorCode>: <Message>”. Again, this is meant at being reasonably *human-friendly* for errors that might get bubbled up to the user via some notification UI.

- The **`debugInfo`** property is the developer’s friend – but it is not exposed by default, which is why you need to explicitly log it. This hidden debug info object will, when possible, include information about what method or property caused the error to occur. This is why you’ll often see the multi-line `catch`, as in the “*Canonical code sample*” at the beginning of this chapter.

  In real-world scenarios, you will likely want to create your own error-handling function, both for convenience and re-usability. Within it, you would want to expose the high-level summary (i.e., `toString`) of the error

[7] In practice, there are a number of errors that do sound rather technical: for example, “***RunMustReturnPromise:** The batch function passed to the “.run” method didn’t return a promise. {…}”. However, these more-technical errors are ones where there’s clearly a coding error on the developer’s behalf, and that would get caught early in the development phase.

to the user, typically via a notification toast or dialog. You would then do the additional logging, making use of the `debugInfo`, as follows:

```
    console.log(error);
    // Log additional information, if applicable:
    if (error instanceof OfficeExtension.Error) {
        console.log(error.debugInfo);
    }
```

Once you have your error-handling function, you would use it as follows:

Using the custom error handler via a ‘.catch‘ statement

```
    Word.run(async (context) => {
        // ...
        await context.sync();
    }).catch(customErrorHandler);
```

... or ...

Using the custom error handler via a ‘try/catch‘ and an ‘await‘

```
    try {
        await Word.run(async (context) => {
            // ...
            await context.sync();
        });
    }
    catch (e) {
        customErrorHandler(e);
    }
```

- Finally, the error object may include **`traceMessages`**, which are custom messages that you can log yourself. See “*Finding the point of failure: Trace Messages*” for more information on using trace messages.

BEST PRACTICE: Always (ALWAYS!) include a `.catch` statement (or `catch` keyword, with an appropriate `await` in front of a Promise!) in the outermost caller of the function. Without `.catch`, if an error bubbles up the call hierarchy and still doesn't find a `.catch` at the top, it simply gets silently ignored – and becomes very hard to spot!

For maximum effect, the `.catch` should be at the outermost layer of the call stack. This means that you should place the `.catch` ***outside*** of the `Word.run` (or `Excel.run`, etc). This ensures that *all* errors get caught.

If you were to place the `.catch` *within* the `Excel.run` method, as part of the returned Promise, the `.catch` would miss errors that *happen before the first* `sync`, or that happen at the end of the batch, if you forgot a final `.then(context.sync)`. I've had this happen to me several times in the past, with code stopping midway through the operation, and yet my `.catch` breakpoint not triggering. So, be aware, and always include your `.catch` at the end.

5.4.2 Don't forget the user!

In most samples in this book – and in the official documentation – you will see that the `.catch` method uses `console.log`.

While this is a fine thing to do, remember that the user won't see these error notifications (which, for a JSON-ified `debugInfo`, may very well be a good thing) – but it also means that the user might not realize that the function has errored out and stopped processing. By the same token, when running/debugging your own code, the errors may also appear silent to you, unless you're looking in the output window!

This is yet another reason to create your own error-handling function, and have it display the errors on the UI. I recommend *always* including the error-handling function, irrespective of how simple the code might be, as there might still be some corner-cases or issues beyond your control, where the user should have at least some indication of the failure.

Note that in the built-in Office Add-in project template that comes with Visual Studio, you can use "`app.showNotification(header, text)`" as a simple way of displaying these errors in a taskpane or content add-in. Alternatively, if you are using Office JS Helpers (more on that below), you can use "`OfficeHelpers.UI.notify(title, text)`". For functions that run *headless* as a result of a ribbon-button click (known more formally as "Add-in commands"), you will want to pop out a standalone dialog. See section "*Pop-up dialogs*" for instructions.

5.4.3 Some practical advice

Personally, I have adopted the following patterns for my error-handling (which you can also see in most of my snippets, and some in the general Script Lab snippets as well)

1. Create a helper function (I call mine `tryCatch`), which takes in a `callback` parameter (which is the passed-in code that it will ultimately invoke).
2. Have the helper function both log to the console, and show some UI. Personally, I highly recommend the `OfficeHelpers.UI.notify` and the `OfficeHelpers.Utilities.log` function, at least to get you started (you can always create your own variant)
3. Write your code in normal standalone functions, always returning a Promise (or just making sure to `await` the `Excel.run` or `Word.run` code – the `async` keyword automatically creates a Promise on your behalf)
4. In the outermost *caller* to your function (e.g., when defining a button-click handler), register the `tryCatch` helper, passing in your actual function as the callback)

Regarding the `OfficeHelpers` functionality, the code is available from the "*Office JavaScript API Helpers*" project (see https://github.com/OfficeDev/office-js-helpers, or via the `office-js-helpers` NPM package).

Putting this advice together leads to the following code (which coincidentally is just the "Basic API Call" sample in Script Lab):

```
$("#run").click(() => tryCatch(run));

async function run() {
    await Excel.run(async (context) => {
        const range = context.workbook.getSelectedRange();
        range.format.fill.color = "yellow";
        range.load("address");

        await context.sync()

        console.log(`The range address was "${range.address}".`);
    });
}

/** Default helper for invoking an action and handling errors. */
async function tryCatch(callback) {
    try {
        await callback();
    }
    catch (error) {
        OfficeHelpers.UI.notify(error);
        OfficeHelpers.Utilities.log(error);
    }
}
```

5.5 Recap: the four basic principles of Office.js

While Office.js may initially seem like unfamiliar terriotory, I firmly believe that its basic principles are very much teachable. With a bit of patience, the proxy-object and `load`/`sync` concepts will become second-nature. In fact, the entirety of the core Office.js topics can be summarized in just a few bullet points!

1. **Any API object is a proxy object**, representing an element that might or might not exist in the document. **Any method invocations**, or any properties that are set, get added to an internal queue, but **aren't dispatched until you call an explicit `await context.sync()`**.
2. **The API is fundamentally batch-centric**. You can queue up as many operations as you want in one `await context.sync()` (which serves as the boundary marker for each batch). You can even call methods on objects that don't exist yet, as long as those objects will come into existence as part of executing the batch (e.g., setting values on a range for a sheet that doesn't exist yet, but that is queued to be added as part of a preceding call).
3. **If you want to *read* document properties, you need to *declare your intention to read them*** before you actually read the data. This is necessary because the JavaScript runtime has no knowledge of the Office document, which sits on an entirely different process/machine, and would require an asynchronous call to get to. That being said, understanding and queueing up requests to read data is pretty easy, if you understand the concept and follow a simple process. The easiest way is to write the code *as if the properties are already loaded* (and using IntelliSense to guide you). Then, go up one or more lines, and add a `load` statement specifying the property name that you're using (i.e., `range.load("value")`), as well an `await context.sync()`. You can queue up `load` statements on as many objects as you wish as part of a single `await context.sync()`. You just need to make sure that both a `load` and an `await context.sync()` happen somewhere upstream of where you are using a given property.
4. For perf's sake, **the add-in developer's job is to minimize `await context.sync()` calls**. Typically, dispatching a bunch of `load` statements in the first batch (for data that you need to read, before you can operate

further), followed by a bunch of automation actions to manipulate the document in the second batch, is a very efficient way of minimizing `await context.sync()` calls to just 2 or 3 for most scenarios.

Here’s a code block that demonstrates the entirety of these four principles:

```
await Excel.run(async context => {
    let sheet = context.workbook.worksheets.getItem("Sales");
    let taxNamedRange = sheet.getRange("TaxRate");
    taxNamedRange.load("values");
    await context.sync();
    if (taxNamedRange[0][0] >= 0) {
        taxNamedRange.format.fill.color = "green";
    } else {
        OfficeHelpers.UI.notify("Tax rate may not be negative!");
        taxNamedRange.format.fill.color = "red";
    }
    await context.sync();
});
```

Try it out

Just import the following Snippet ID into Script Lab:
39c81dabec330a70e72e194b06c36bce

6. Implementation details, if you want to know how it *really* works

In writing this book and receiving feedback from early readers, I've heard several requests for a more thorough explanation of what happens under the covers with all this proxy-object / syncing business.

So, if you're the sort of person who likes to see the **implementation details** in order to better grasp a technology – and if curiosity about the internal workings is going to distract you until you know the answer – let's pause here and let you read on for a moment.

If you're not, feel free to skip to the next chapter. You can always return here later, after you've seen more of the APIs in action, if *that's* when your curiosity awakens.

Chapter structure

6.1 The Request Context queue

At the heart of the new wave of Office 2016 APIs is a Request Context – which is the object you receive as a parameter to the batch function, inside of an `Excel.run`. Fundamentally, you can think of a Request Context object as a central repository that accumulates any changes you'd like to do to the document. I say "repository", because the Request Context could indeed be likened to a version-control system, where all you send is the *diff*-s between the local state and the remote state[1].

The Request Context holds two arrays that allow it do it its work. One is for **object paths**: descriptions of how to derive one object from another (i.e., "*call method* `getRow` *with parameter value* `2` *on <insert-some-preceding-object-path> in order to derive this object*"). The other is for **actions** (i.e., *set the property named "color" to a value of "purple" on the object described by object path #xyz*). For those who are familiar with the "Command" Design Pattern, this notion of carrying around objects that represent the recipe for a particular action should sound quite familiar.

On the Request Context is a single root object that connects it to the underlying object model. For Excel, this object is a `workbook`; for Word, it is `document`. From there, you can derive new objects by calling methods on that root proxy object, or on any of its descendants. For example, to get a worksheet named "Report", you would ask the `workbook` object for its `worksheets` property (which returns a proxy object corresponding to the worksheets collection in the document), and then use `worksheets` to call a `getItem("Report")` method in order to get a proxy object corresponding to the desired "Report" worksheet. Each of these objects carries a link to its original Request Context, which in turn keeps track of each object's path info: namely, who was the parent of this new object, and what were the circumstances under which it got created (*was it a property or a method-call? were there any parameters passed in?*).

Whenever a method or property gets called on a proxy object, the call is

[1] Git is a particularly well-suited version-control analogy, as local changes are so perfectly isolated from the repository: until you do a `git push` of your local state, the repository has *no knowledge whatsoever* about the changes being made! The Request Context and proxy objects of the new Office.js model are very much the same: they are completely unknown to the document until the developer issues a `context.sync()` command!

registered as an **action** on the Request Context[2]. For example, a call to `range.merge()` or the setting of `fill.color = "purple"`, will get put in the queue as action such-and-such on object such-and-such. Moreover, if the result of the method or property call is another proxy object (for example, `worksheets.getItem("Report")` or `worksheets.add()`, a new proxy object will be generated as a *side-effect* of the method call, and its lineage will be dutifully noted by the omniscient Request Context.

Let's walk through a real example. Suppose you have the following code:

Tracing through the Request Context / proxy object operations

```
1       Excel.run(async (context) => {
2           let range = context.workbook.getSelectedRange();
3           range.clear();
4           let thirdRow = range.getRow(2);
5           thirdRow.format.fill.color = "purple";
6   
7           await context.sync();
8       }).catch(OfficeHelpers.Utilities.log);
```

Now let's analyze it with a fine-toothed comb. With each API call, I will keep a running tally of the object path and their actions (expressed in a friend-liefied and shortened notation, but following closely to what happens internally).

To start, line **#1** – `Excel.run(async (context) => {` – uses `Excel.run` to create a Request Context object. The `.run` invocation does a number of other things too (See "`Excel.run` *(*`Word.run`*, etc.) advanced topics*"), but let's leave it be for the time being. The important thing is that it gives us a brand new `context` object, on which there is already a pre-initialized a `workbook` object (which we'll use momentarily).

[2] The property-setting is intercepted by subscribing to the setter for the property – see TypeScript getters and setters, which ultimately translate to `object.defineProperty` in the EcmaScript 5 (ES5) incarnation of JavaScript.

```
    objectPaths:
        1 => global object (workbook)

    actions: <none>,
```

On line **#2** – `let range = context.workbook.getSelectedRange()` – we use that `workbook` object to derive a new object, corresponding to the current selection. We assign it to a variable called `range`, but it doesn't matter to the Request Context: even if we hadn't named it anything, and instead used it as a pass-through object to get to another destination, it would still get reflected in the Request Context's list. The creation of the object also gets reflected as an object-initialization action in the actions list, for purposes described later in this section.

```
    objectPaths:
        P1 => global object (workbook)
        P2 => (range)
                parent: "P1", type: "method",
                name: "getSelectedRange", args: <none>

    actions:
        A1 => action: "init", object: "P2" (range)
```

Line **#3** – `range.clear()` – adds the first real document-impacting action: a command to clear the contents of the range:

```
    objectPaths:
        P1 => global object (workbook)
        P2 => (range)
                parent: "P1", type: "method",
                name: "getSelectedRange", args: <none>

    actions:
        A1 => action: "init", object: "P2" (range)
        A2 => action: "method", object: "P2" (range)
                name: "clear", args: <none>
```

Line **#4** – `let thirdRow = range.getRow(2)` – follows a similar pattern as line #2, creating a `thirdRow` object that derives from the previously-defined `range` object, and adding another instantiation action:

```
    objectPaths:
        P1 => global object (workbook)
        P2 => (range)
                parent: "P1", type: "method",
                name: "getSelectedRange", args: <none>
        P3 => (thirdRow)
                parent: "P2", type: "method",
                name: "getRow", args: [2]

    actions:
        A1 => action: "init", object: "P2" (range)
        A2 => action: "method", object: "P2" (range)
                name: "clear", args: <none>
        A3 => action: "init", object: "P3" (thirdRow)
```

Line **#5** – `thirdRow.format.fill.color = "purple"` – is packed with several API calls. We begin with creating an [anonymous] format object, by following the `format` property of the `thirdRow` variable. We then do the same for the [anonymous] fill object. Both follow the same pattern as before, creating an object path and an instantiation action for each. But then, having reached the desired object, we do another document-impacting action on the object: setting the fill color of the third row to purple (see action "**A6**" below):

```
    objectPaths:
        P1 => global object (workbook)
        P2 => (range)
                parent: "P1", type: "method",
                name: "getSelectedRange", args: <none>
        P3 => (thirdRow)
                parent: "P2", type: "method",
                name: "getRow", args: [2]
        P4 => (format)
                parent: "P3", type: "property",
                name: "format"
        P5 => (fill)
                parent: "P4", type: "property",
                name: "fill"

    actions:
        A1 => action: "init", object: "P2" (range)
        A2 => action: "method", object: "P2" (range)
                name: "clear", args: <none>
        A3 => action: "init", object: "P3" (thirdRow)
        A4 => action: "init", object: "P4" (format)
        A5 => action: "init", object: "P5" (fill)
        A6 => action: "setter", object: "P5" (fill),
                name: "color", value: "purple"
```

And finally, on line #7 (line #6 was a blank one), we get to the magic **`await context.sync()`** incantation. This command tells the Request Context to pack up all of the relevant information (namely, pending actions, and any associated object path info-s), and send it to the host application for processing.

On the receiving ends of the host application, the host unpacks the actions and begins iterating through them one-by-one. It keeps a working dictionary of the objects that got derived during this particular `sync` session – such that, having retrieved the range corresponding to `thirdRow` once, it will not need to re-evaluate it again. This is done not only for efficiency, but also to prevent mistakes: you wouldn't want to re-fetch the row at relative index 2 if another few rows got added between it and the first row; nor would you want to re-fetch the selection every time, since it may well have shifted (e.g., during the adding and activating of a new worksheet), and yet semantically the range should be *imprinted* with the original reference. Finally, if you have an object derived from calling the `add` method on the worksheet-collection object, you *definitely* wouldn't want to re-derive the object – and, as a side effect, add a new sheet – every time that the object was accessed!

If at any point in the chain something goes wrong, the rest of the batch gets aborted. Going with the previous example, if there is no third row in the selection (i.e., it's a 2x2 cell selection), the remaining commands would get ignored (which is probably what you'd expect, anyway). Importantly, though, there is no *atomicity* to the `Excel.run` or the `sync`: any actions that have already been done will *stay* done. In the case of this example, the document might be left in a *dirtied* state, where the clearing of the selection has already happened, but the formatting of the third row has not been done yet. While not ideal, this is no different from VBA or VSTO with regards to Office automation; it is simply too difficult to roll back, especially given any user or collaborator actions that may have happened in the meantime[3]. See the section "***Handling errors***" for my recommendations on how to minimize the risk of leaving a user in an undesired state.

6.2 The host application's response

Let's assume that the sync did succeed: that every necessary object (the original selection, its third row, the format, the fill) all were created successfully, and that both document-impacting actions were also able to commit to the document. What happens next?

As mentioned earlier, the host keeps a running dictionary of the object that it's been working with. But this running dictionary is *only for the duration of the particular* sync*: not for the lifetime of the application*. To keep and track the objects indefinitely would be a huge performance hit.

Now, let's take the case where an object path is the "add" action on a worksheet collection. During the processing of the sync, the method would only have been executed once (with the appropriate side-effect of creating the worksheet), and the resulting sheet would be cached. This is great for the current sync, but what if the developer wants to access the sheet again at a later sync? This is where the instantiation actions mentioned earlier come in.

For each action, the host application may *optionally* send a response. For actions like clearing a range or setting a fill color, there is nothing to respond with (the fact that the operation succeeded is obvious through the fact that the queue kept executing to completion). But for instantiation actions, the host *may* send a response to tell JavaScript to re-map its object path to something less volatile. Thus, while the original path for a newly-created sheet may

[3] While atomicity is possible in well-structured, GUIDs-everywhere databases, it's much harder to imagine it in the loose structure of an Excel or Word document. I should also note that in Excel (but not Word), you will see that any document-impacting (write) operations will cause the undo stack to be blown away – again, far from ideal, but no different than VBA's or VSTO's behavior today for Excel.

have been "*execute method '`add` on object xyz*" (where xyz is the worksheet collection), the response might indicate "*from here on out, refer to the sheet as being a "getItem" invocation with parameter "123456789" on that same xyz object". That is, in creating the object and executing the instantiation action, the host can figure out if there is a more permanent ID it can give back to JavaScript for future references to this object. (A less drastic example: fetching a sheet by name is somewhat risky, in that names can change, both via user interaction and programmatically; but if the host can re-map the path to a permanent worksheet ID, any future invocations on the object are guaranteed to continue to refer to the same sheet, regardless of its name).

But there is another, even more important use for responses from the host. Suppose, on the JavaScript side, you have a call to `range.load("formulas")`. In terms of actions, this gets represented as a *query* action on the object, with a parameter whose value is "formulas". To this action, the host will respond by fetching the appropriate object (which is already in its dictionary, thanks to the instantiation action), querying it for the required properties, and returning the requested information.

6.3 Back on the proxy object's territory

Back in JavaScript, the `sync` is patiently waiting for a response from the host application. And, hopefully, the developer's code is *also* patiently waiting, by either using an `await` or subscribing to the `.then` function-call of the `sync` Promise.

When the response *does* come back, there is a bit of internal processing before the execution gets back to the developer's code. For example, any of the path-remappings, described in the preceding section, take effect. There is also some internal processing (e.g., invalidating the paths of objects that were valid during the previous `sync` batch, but cannot be used again – I'll explain more soon). And, importantly, the results of any *query* actions take effect, taking the loaded values and putting them back on the corresponding objects and properties. This ensures that, following the `sync` if the developer's code now references `range.values` for a Range whose values have been loaded, he/she will get the last known snapshot of the values (as opposed to a `PropertyNotLoaded` error, mentioned earlier in "*Loading properties: the bare basics*").

With the post-processing done, the request context is now ready to be re-used. Its actions array was reset to a blank slate at the very beginning of the `sync`; and conversely, the object paths array (which is never emptied during the lifetime of the particular request context, as later actions are bound to re-use some of the existing paths) has had any of the object paths tweaked, based on responses from the host and post-processing. And so a new batch of operations can begin, queuing up until the next `await context.sync()`.

6.4 A special (but common) case: objects without IDs

When working with objects like worksheets (Excel) or content controls (Word), the host application's job is quite easy: in both cases, there is a permanent ID attached to each of those objects, so no matter how the object was created (`getActiveWorksheet()`, or `getItem`, or whatever other invocation), the host can always use the instantiation action to re-map the path back to a permanent[4] ID. Which means that, as a developer, having created the object once at some point, you can continue to use it in the next `sync`, or even longer thereafter[5]. No surprises there.

But what about objects that don't have IDs; and that, by definition, have an infinite number of permutations about them? Both Excel ranges (a particular grouping of cells) and Word ranges (some text starting at one location and ending in another) are not at all easy to get a concrete reference to: the address/index at which they are might shift, and the ranges might also grow and expand if additional cells or characters are added within them. The same

[4] A "permanent ID", in this context, means an ID that won't change for the *session* of the document being currently opened. The IDs are not necessarily guaranteed to remain the same during a re-opening of the document, so you should check the documentation carefully for whether an ID can be used across document-open sessions. But for purposes of the API processing and the JavaScript (which also is only open for the duration of the document, or shorter), the ID is *permanent enough*.

[5] Such objects can even be used outside of the current `Excel.run`, in a procedure some time later (e.g., as part of a subsequent user button-click). See http://stackoverflow.com/a/37290133/678505, and note that an even better approach is forthcoming.

is true for some other objects[6]

The host doesn't have issues tracking the objects during the *processing* of the batch, as there it can use an internal dictionary and, having retrieved the object once, continue to cling on to it for the duration of the batch. But as noted earlier, the host is *stateless* across batches – it cannot afford to keep a reference to every object ever accessed on it from JavaScript, or else the application will leak memory and crawl to a halt.

To avoid bogging down the host application with thousands of no-longer-needed objects, while still enabling the very common scenario of needing to do a `load` and a `sync` before performing further actions with an object, the original design was as follows:

1. By default, any object without an ID quietly loses connection with the underlying document, and cannot be used again after the current `sync`. It can still be read-from by JavaScript *after* the sync (otherwise, there would be no point to loading it in the first place!), and that might be fine for some scenarios – but this default behavior would preclude, for example, the highlighting example where the range values are first read, and then colored accordingly.
2. To enable the latter scenario, we expose an API – `context.trackedObjects.add` – where a developer can explicitly tell the host that "*I want to track this object longer-term, not just for the current* `sync`. The developer is then responsible for calling `context.trackedObjects.remove` when the object is no longer needed (no hard penalty if he/she doesn't, but it will slow down the host application over time, so use object-tracking sparingly, and try to release as soon as you're done).

On the JavaScript layer, a call to `context.trackedObjects.add` would add a new type of action to the queue, saying that object with id such-and-such would

[6] For the sake of painting a complete picture, one clarification: the problem isn't so much objects that don't have IDs, but rather ones that can't be *re-derived*. For example, a Range object – whether created via address, or from a selection, or intersection, or whatever else – cannot be *re-derived* in a later batch, because it may well have shifted in the meantime, and so re-evaluating the object path might lead to a completely different object. On the other hand, an Excel `Format` object obtained from `range.format`, or a Fill object obtained from `format.fill` does *not* need to have an ID (it doesn't) *nor* be tracked, because as long as the parent range remains tracked and valid, the format or fill can *un-ambiguously* be brought back to life.

like to be tracked. On the host side, this action would be interpreted to create a permanent wrapper around the in-memory object, creating a made-up ID that the object could use as if it were a real ID. This ID would be sent back to the object, very much like the object-path-remapping result of an instantiation action. And likewise, a call to `context.trackedObjects.remove` would likewise get a special action added on the queue, requesting that the host release the memory for the no-longer-needed object, and marking the object itself as no longer having a valid path.

This design worked – and in fact, it still works today, if a developer chooses to create a Request Context manually, via `var context = new Excel.RequestContext()`, instead of a `.run`. But in practice, in both our internal testing and public preview, it tuned out to be very tedious to have to call `context.trackedObjects.add` on an object or two within nearly every scenario. And even when developers did call it (with some trial-and-error), it was even more tedious (nay, unrealistic) to expect that folks will remember and correctly dispose of the no-longer-needed tracked objects.

In observing folks struggle with this tracked-objects concept, one thing that became clear is that in the vast majority of cases, the developer's intent is *not* to keep the object around for some long-term storage – rather, the developer generally just needs to track the object so that they can use it across one or two `sync` boundaries, and then they are done with it forever. This is where the `Excel.run` (`Word.run`, etc.) concept was born: to allow developers to declare a single semantic unit of automation, even if internally it is comprised of a series of `sync`-s. And for the framework to handle the tracking and untracking silently.

This means that whenever you do an `Excel.run` (`Word.run`, etc.), after each instantiation action there is *also* an action to track the object. And at the very end, after a final flush of the queue at the completion of the `Excel.run`, there is a separate internal request made to un-track every non-ID-able and non-derivable object that had been created in the meantime. So the *true* picture of the "actions" array from above is actually a bit more verbose than shown earlier:

```
    actions:
        A1 => action: "init", object: "P2" (range)
        A2 => action: "track", object: "P2" (range)
        A3 => action: "method", object: "P2" (range)
                name: "clear", args: <none>
        A4 => action: "init", object: "P3" (thirdRow)
        A5 => action: "track", object: "P3" (thirdRow)
        A6 => action: "init", object: "P4" (format)
        A7 => action: "init", object: "P5" (fill)
        A8 => action: "setter", object: "P5" (fill),
                name: "color", value: "purple"
```

And then, at the completion of the `run`, after a final flush of the queue, the following would be sent (note that only the relevant object paths are being sent; there is no need to carry extra baggage over the wire/process-boundary):

```
    objectPaths:
        P1 => global object (workbook)
        P2 => (range)
                parent: "P1", type: "method",
                name: "getSelectedRange", args: <none>
        P3 => (thirdRow)
                parent: "P2", type: "method",
                name: "getRow", args: [2]

    actions:
        A1 => action: "untrack", object: "P2" (range)
        A2 => action: "untrack", object: "P3" (thirdRow)
```

And this – in a not so small nutshell – is how the underlying proxy objects work, and how the runtime handles its communication to and from the host application.

7. More core `load` concepts

Chapter structure

- Scalar vs. navigation properties – and their impact on `load`
- Loading and re-loading
- Loading collections
- Understanding the `PropertyNotLoaded` error
    - What does it even mean?
    - Signaling your intentions: an analogy
    - Common mistake: re-invoking `get` methods (and in general, loading on methods versus properties)]
    - Another common mistake: loading nonexistent properties
- Methods that return “primitive” types (strings, numbers, etc). E.g.: tables.getCount(), chart.getImage(), etc.

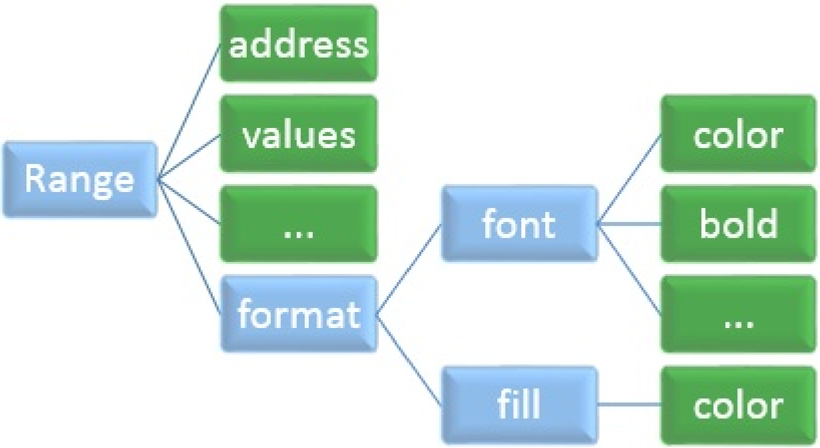
7.1 Scalar vs. navigation properties – and their impact on `load`

A bit of terminology: simple properties like `values` or `numberFormat` off of an API object like `Excel.Range` are called **scalar properties**. These **scalar properties** are either *primitive types* (numbers, Booleans, strings), *arrays of primitive types*, or *complex objects* (i.e., plain JS objects composed of primitives or arrays of primitives; the equivalent of a C++/C# struct). One way to think about scalar properties is that they are the sort of simple properties you could serialize and send over the wire via JSON, XML, etc.

By contrast, any API objects are instances of API classes, and are not serializable (they are classes, with methods and all, not just pure data!). Such properties are called "**navigation properties**", because they're used to navigate you from one API object to another – but they ultimately carry no data of their own. So, whereas you need to load the scalar properties that you intend to use, **you do **not** need to load navigation properties**.

In case an analogy helps, you can imagine the object hierarchy as a tree structure. In this case, let's imagine we start with the `Range` object.

- Any ***leaf node*** properties – whether on the `Range` object directly, or on one of its descendants – are **scalar properties** (marked in green).
- Any ***non-scalars*** are **navigation properties** (marked in blue).


Range
address
values
...
format
font
fill
color
bold
...
color

KNOWLEDGE CHECK:

Which of the following are scalar properties?

1. **Excel APIs**: `range.address`
   returns a string of the form
   `Sheet1!A1:B5`

2. **Word APIs**: `body.text`
   returns a string of the document body's text
   "Lorem ipsum dolor sit amet, consectetur adipiscing elit."

3. **Excel APIs**: `range.values`
   return an array of string/number/Boolean values, e.g.,
   [["Item", "Price"], ["Hammer", 17.99]]

4. **Excel APIs**: `table.sort.fields`
   returns an array of SortField interfaces (pure JS objects). e.g.,
   `[`
   ' { key: 0, sortOn: value, ascending: true }, { key: 1, sortOn: value, ascending: true }]'

5. **Word APIs**: `paragraph.font` (or Excel's `range.format.font`)
   returns an API object from which you can access the properties `bold`, `color`, `size`, *etc.*
   `var font = paragraph.font;`

6. **Word APIs**: `body.paragraphs`
   returns a collection of Paragraph API objects
   `var paragraphs = body.paragraphs;`

Answers:

- ***#1-3* are all scalars**, because they are all either *primitive* types (strings/Booleans/numbers), or arrays of primitive types. In short, they are just regular properties, not API objects.

- ***#4* is *also* scalar**, because it returns an array of objects that, while not *primitive*, are still pure JS objects; they are *NOT* instances of Excel API object classes (i.e., they do not derive from the `OfficeExtension.ClientObject` class, or any class at all!).

  In TypeScript IntelliSense, you'll see that the type of the result is `Excel.SortField[]`, but `SortField` is just a TypeScript interface designed to give a formal structure description to a JS object – it does not imply that the object is an API object!

- ***#5* is *NOT* scalar**, because "font" *is* a real API object. How can you tell? First, it has a `load` method, and a `context` property, both of which are signs of deriving from the `OfficeExtension.ClientObject` class. Second: its properties can be individually set (as opposed to a complex object like `SortField`, which would have needed to be set all in one go). Thus, while `font` doesn't necessarily feel like a *real* document object, it is no different from a `Range` or `Worksheet` or `Paragraph` from a loading perspective: it is a **proxy object**, not a scalar type.

- ***#6* is *NOT* scalar**, because **a collection is *also* an Office API object** (and a very special one, at that). A collection of API objects is very different from an array of scalar types.

When specifying properties to load, you can specify either scalar properties either directly off the object, or scalar properties that are accessible via navigation properties off the object. For example, to load both the cell address and the fill color of a Range, you would specify the following (note that the scalar property is always the "leaf-node" one – and any preceding slashes indicate navigation properties):

```
myRange.load("address, format/fill/color");
```

Technically speaking, you are not required to include the property names. **You could (though you shouldn't!) write a `load` statement with no property names passed in** (e.g., `myRange.load()` or `context.load(myRange)`). But if you do that, Office.js will **load all scalar properties** on the object – which on the Excel Range object, for example, is over a dozen! – even if you end up using only one of them! Moreover:

1. For some properties (i.e., Range values in Excel), loading a particular property might fail, even if some of the other properties would load just fine. Concrete example: an unbounded range like columns A:C will throw an error if you try to load their `.values` property (that's upwards of a million cells in each column, and so the API blocks unbounded ranges!), but would respond just fine to the `address` property.
2. Even if you don't notice the difference on Desktop, the delay (and bandwidth costs!) of the extra properties will almost certainly show in Office Online.
3. Even if an object exposes just two properties today, and you're using them both, that's not to say that more won't be exposed eventually! This means that your add-in might slow down (and start chewing through more bandwidth) over time, as more APIs are added. In short, even if you're currently using all properties of the object, and hence don't see a need to load the properties explicitly, you are still better off writing them out to prevent future slowdown.
4. The effects of #2 and #3 are multiplied ten-fold or hundred-fold when loading a collection (proportional to the number of elements in the collection – which often-times is something that the user controls, but not the developer!)

BEST PRACTICE:

In addition to the “Golden Rule” above, which should prevent you from loading unnecessarily, ***you should also avoid loading unneeded properties***. That is, even if a `load` *is* necessary, be sure that you aren’t loading properties that you’re not using.

Be particularly wary of the “*empty load*” mentioned earlier – that is, a load that does *not* call out the properties explicitly (for example, `range.load()`). Such `load` statement is the equivalent of loading *all* of the *scalar* properties on the object (that is, for an `Excel.Range` object, it will load formulas, values, number formats, address, row count, column count, etc). The situation is *even worse for collections*, where an “*empty load*” will also load all the scalar properties on the collection’s *children*, as well. So, for example, an innocent-looking Word invocation of `document.body.paragraphs.load()` will load the alignment, indents, line spacings, spacings before and after, and of course the full length texts *of every single paragraph in the document*!

The reason it’s called an “*empty load*” is because the `properties` parameter to the `load` function is missing (empty). But in terms of data traveling over the wire, it’s very much the opposite of “empty”. Also, note that the alternate syntax of `context.load(range)` still constitutes an *empty load* because the `load`-function off of the context object expects two parameters, where the latter is the property(ies) to load.

7.2 Loading and re-loading

If you receive a **`PropertyNotLoaded` error** when accessing a property, you must have forgotten to load it! The fix is fortunately very simple: add the `load`, and ensure there is a `sync` somewhere upstream from where you're using the property.

Note that the `PropertyNotLoaded` error will ***only*** be thrown when you ***initially*** **forget to load the property**. If you manipulate the object (i.e., *set the* `formula` *property on a cell where you've previously loaded a* `value`), or if it gets impacted by other external factors (*i.e.: a formula dependency on another cell*), **it is your responsibility to remember to re-load the property:**

Re-loading properties

```
// ...
// Initial loading of values:
myRange.load("values");

return context.sync()
    .then(function() {
        // ... Some operations that impact range values

        // Re-load the cell values to retrieve the latest:
        myRange.load("values");
    });
// ....
```

When loading or re-loading properties on a *regular (non-collection*[1]*)* object, note that **only the specified property names** will be re-loaded; the rest will be kept as is. So, for example, loading `myRange.load("address")` within the `.then` above would ***not*** have refreshed the `values` property!

[1] For re-loading on collections instead of regular API objects, see the end of section "*Loading collections*". Essentially, re-loading a collection *blows away any existing objects and their properties*, so you have to load *everything* from scratch – you can't just additively load a couple of new properties while maintaining the existing items.

7.3 Loading collections

Loading collections is similar to loading regular objects – except that the properties you specify are the properties of the *children*. There are a few nuances to collections in particular (see "*Loading collections*" for more details), but the general use is easy enough. Simply call `load` on the collection, passing in the **names of the *child* properties**. After syncing, you can **access the items using the collection's `items` property**. For example, to list out the names of all worksheets in Excel, you would do:

Loading properties on a collection

```
Excel.run(async (context) => {
    let sheets = context.workbook.worksheets;
    sheets.load("name");

    await context.sync();

    for (var i = 0; i < sheets.items.length; i++) {
        console.log(sheets.items[i].name)
    }

}).catch(...);
```

Again, notice that with a collection:

- When loading, you specify the **name(s) of the child property(ies)**, not properties of the collection itself (of which there are usually very few, typically just a `.count`, if that).
- When accessing loaded items, you **use the `.items` property** on the collection. The `items` property returns a plain 0-indexed JS array containing the loaded elements (with the specified properties pre-loaded).
- If you find it unsymmetrical to load the `name` child property, but use it as `items[x].name`, you can specify the load statement as `load("items/name")` instead. Internally, when the runtime sees this syntax, it simply strip out the `items/` portion of the load statement. It's up to you which syntax

you wish to use[2], but know that the *canonical* load statement is just `load("name")`.

Common error when accessing items on a collection:

It is a common mistake to accidentally omit the `.items` portion, when reading back collection elements. That is, instead of **sheets.items[i]**`.name`, trying to access the property as just **sheets[i]**`.name`.

This code will throw a runtime error, "*TypeError: Unable to get property 'name' of undefined or null reference*". Why? Because while it's true that `sheet.items` is a plain JS array – with the ability to access items in it via an index, to call `.forEach` on the array, etc., – `sheets` itself is NOT an array, but rather an API collection object! So, please remember not to omit the `.items` portion when accessing loaded items on a collection!

You can load more than just one level deep on a collection. For example, to enumerate all the column names on all the tables on all worksheets on a workbook, you could do:

A multi-level load

```
Excel.run(async (context) => {
    workbook.load("worksheets/tables/columns/name");

    await context.sync();

    var secondColumnOfFirstTableOnThirdSheet =
        workbook
        .worksheets.items[2]
        .tables.items[0]
        .columns.items[1];
```

[2] As of the time of this writing (early November 2016), one unfortunate limitation is that the use of the `items` prefix (i.e., `load("items/name")`) is only available in `ExcelApi 1.2+` and `WordApi 1.2+` on the Desktop – but not in the `1.1` API Sets for these two hosts. However, it might be possible that this limitation will be lifted… stay tuned.

```

    console.log("The ridiculously-nested column name was " +
        secondColumnOfFirstTableOnThirdSheet.name);

}).catch(...);
```

Note that above, the collection-item access has `.items` at each level in the hierarchy:

`worksheets.`**items[x].**`tables.`**items[y].**`columns.`**items[z].**`name`

As mentioned above, if it's easier for you to think of it in this structure when loading as well, feel free to just add "`items/`" at each layer in your load statement (which the runtime will then strip out, bringing the `load` statement back to its *canonical* form[3]):

`workbook.load("worksheets/`**items/**`tables/`**items/**`columns/`**items/**`name");`

Finally, it is worth re-iterating that **unlike with regular proxy objects**, you **must** load all the properties that you want on the collection **at once**, via a list of comma-separated property names:

Loading multiple properties on the children of the collection

```
Excel.run(async (context) => {
    let worksheets = context.workbook.worksheets;
    worksheets.load("name, visibility");

    await context.sync();

    console.log("Visible sheets are: ")

    worksheets.items
        .filter(sheet =>
            sheet.visibility === Excel.SheetVisibility.visible
        )
        .forEach(sheet =>
```

[3] Again, beware of that limitation that currently, the "`items/`" syntax is only allowable on `ExcelApi 1.2+` or `WordApi 1.2+` on the Desktop – but not on the `1.1` release. See the previous note for more details.

```
            console.log("- " + sheet.name);
        );
})
.catch(...);
```

If the above code had the `load` statement split in two – one for `name`, and the other for `visibility` – this wouldn't just be a perf-hit, like it would for a regular object. Instead, the second call would throw away any previously-loaded items, and so `name` would *no longer be loaded*. And hence, you would get:

> ***PropertyNotLoaded****: The property 'name' is not available. Before reading the property's value, call the load method on the containing object and call* `context.sync()` *on the associated request context.*
>
> *Debug info:* `{"errorLocation": "Worksheet.name"}`

7.4 Understanding the `PropertyNotLoaded` error

Sooner or later – and at first, *sooner* and far more often than you'd like – you will try to use a property, and get a `PropertyNotLoaded` error thrown instead. Let's step through what the error means, and how to avoid it.

7.4.1 What does it even mean?

From the perspective of JavaScript, the actual document data is stored somewhere far far away. It might be on the same device, albeit across a process boundary, or – in the case of Office Online – it might be in a data center hundreds of miles away from you. All that the JavaScript has are proxy objects. So for a simple scenario like applying the fill color of one cell to another – something that in VBA would be a one-liner – in the JS APIs you have to do in three lines instead. Namely, given these two ranges:

```
let source = worksheet.getRange("A1");
let dest = worksheet.getRange("A2");
```

You would need to first issue a command to load the desired property (color), then do a `sync` at some point, and finally use the property's loaded value:

```
source.format.fill.load("color");
await context.sync();
dest.format.fill.color = source.format.fill.color;
```

Let's do another example, this time with the Excel Range's `values` property, and this time multiplying the value by 5 for good measure. The pattern is identical, except that you have to remember that `values` is a 2D array, so you'll need to drill down into first column element of the first row it when multiplying:

```
source.load("values");
await context.sync();
dest.values = [[ source.values[0][0] * 5 ]];
```

Is it overly complicated relative to VBA? **Yes**.

Is it about 3-4 times more verbose? **Yes**. *[Though remember, it also will run on 3-4 times as many endpoints: Office Online, Mac, and iOS, in addition to the Desktop PC. So it's not altogether an unfair trade].*

Will you be able to conquer it? **Yes**. If you keep on reading…

7.4.2 Signaling your intentions: an analogy

To repeat: Unlike in VBA, where the entirety of the document was in-memory, in the same process, at your program's immediate disposal, the JS APIs don't have this luxury. You have to *signal your intentions* to want to read a particular bit of data; you then have to wait for that data to be ferried across (`await context.sync()`), and only then can you use it.

> **An analogy, created on the spur-of-the-moment when explaining this concept to a particularly disgruntled developer:**
>
> Think of it like driving. When you've been cruising merrily in the middle lane of the highway, and notice that your exit is coming up in one mile – you don't just wait for the last minute and slice across the highway to it. Instead, you:
> 1. First, signal your intentions, by putting on the right blinker. Much as the JS world has no concept of what is happening in the document, drivers in the cars beside you have no concept of what is happening in your mind. By signaling your intentions, you make it clear that you intend to switch lanes.
> 2. Look out the side or rear-view mirrors; check your blind spot. In short, `sync` your understanding of the world around you with what is actually out there.
> 3. With the information in hand (`load`-ed and `sync`-ed), you may now act (dispatch an action from your hands to the steering wheel to merge into the desired lane).

Want to take the analogy a few steps further? No problem:

- When `sync`-ing your worldview to that of the positions of the cars around you, you need to `await` the `sync`. Failure to await can lead to unexpected/unpredictable behaviors, and even to a crash... both of a car, and your program!
- Even when you try to follow these steps, there can always be unexpected problems: your mirror might be mis-aligned, not letting you see; or when you go to switch lanes, a deer might appear on the side of the road, forcing you to abort your actions. In short, remember to `try` to perform the action, but to also be prepared to `catch` and gracefully handle any

unexpected surprises. (Stepping away from the analogy for a moment: a public service announcement to also `await` whatever you're doing inside the `try` block – for example, the `Excel.run` – or else the `catch` will do you no good).

7.4.3 Common mistake: re-invoking get methods (and in general, loading on methods versus properties)

When working with API objects, it is crucial to take note whether you are

- **Accessing the object via a method**. For example:
  - worksheets.getItem("Sheet1") or
  - workbook.getSelectedRange() or
  - range.getIntersection(anotherRange)

- **OR, accessing the object via a property**. For example:
  - worksheets.items[0] or
  - chart.title, or
  - range.worksheet, etc.

Notice that for objects accessed via methods, them methods will *almost always* be prefixed with a get – so be on a particular lookout for these.

The reason to care about the distinction is that **method invocations – unlike property access – *always return a new object***! Let me show this via two examples: first the ***incorrect*** usage, and then the ***correct*** one.

***Incorrect** use of 'load' when interacting with methods*

```
Excel.run(async (context) => {
    context.workbook.getSelectedRange().load("address");

    await context.sync();

    // INCORRECT:
    console.log(context.workbook.getSelectedRange().address);
})
.catch(...);
```

The above code will produce the following error:

> ***PropertyNotLoaded:*** *The property ‘address’ is not available. Before reading the property’s value, call the load method on the containing object and call* `context.sync()` *on the associated request context.*
>
> *Debug info:* `{"errorLocation":"Range.address"}`

The reason that this is an error is that you are **retrieving the object anew** the second time when you reference it – and so you get a brand new proxy object, which has no knowledge of the information you loaded on its twin. This, by the way, is a very common mistake when working with collections: to first fetch `collection.getItem("key")` and call `load` on it, then sync, and then re-query `collection.getItem("key")` – with the latter being a brand new copy of the original object, which defeats the purpose of having loaded the item to begin with!

Thus, the **proper** way to use `load` on an object that was retrieved via a method call is to **keep a reference to the variable**, and use *it* when reading back the data, rather than re-fetching the item via the method.

The **proper** use of ‘load‘ with methods

```
Excel.run(async (context) => {
    // Store the range into a variable:
    let range = context.workbook.getSelectedRange();
    range.load("address");

    await context.sync();

    console.log(range.address);
})
.catch(...);
```

Again, it’s important to call out that **the guidance above is *purely* for objects retrieved via a *method call***. In the case of **property access**, you ***don’t*** have to store the intermediate navigation properties (unless you want to): after the initial property invocation, the *exact same instance* of the object will be returned time and time again. For example, the following is perfectly valid.

*Loading data and then re-querying a *navigation property* is perfectly fine, with no need for intermediate variables*

```
Excel.run(async (context) => {
    let sheet = context.workbook.worksheets.getItem("Sheet1");
    let chart = sheet.charts.getItemAt(0);

    // Load the text of the chart title. Note that we
    // aren't creating a local variable for the title
    chart.title.load("text");

    await context.sync();

    // Note the "text" scalar-property access via
    // the ".title" navigation property on the chart.
    // This would *not* have worked if ".title"" were
    // instead a method (a hypothetical "getTitle()").
    console.log(chart.title.text);
})
.catch(...);
```

7.4.4 Another common mistake: loading nonexistent properties

There is one easily-overlooked "**gotcha**" with the `load` method: it does *not* throw an exception, *even if you specify incorrect property names*. Instead, it simply no-ops on names that it doesn't recognize.

So: if you think you've loaded a property, and you know you did a `sync` upstream, and yet you're still getting a "*PropertyNotLoaded*" error, check your spelling! For better or worse, the `load` method always succeeds – even if you pass in bogus property names! If `load` sees properties it doesn't know about, it simply ignores them, causing the property that you *wanted* to stubbornly (and justifiably) insist that it *hasn't* been loaded.

My trick for avoiding this issue is to always write out *what I want to access* first, and then fill in an appropriate `load` and `sync` above. That is, if I want to read the values of a range, I will first create a reference to the range object, then go down a few lines, and write out the code that accesses the values (using IntelliSense to guide me on the accessor):

```
let sheet = context.workbook.worksheets.getItem("Sheet1");
let range = sheet.getRange("A5");

// ... skip some lines

if (range.values[0][0] === "") {
    /* ... */
}
```

With this code written, I will literally copy-paste the accessor portion into the argument for a `load` statement (and also add a `sync` somewhere in-between):

```
    let sheet = context.workbook.worksheets.getItem("Sheet1");
    let range = sheet.getRange("A5");

    range.load("values");
    await context.sync();

    if (range.values[0][0] === "") {
        /* ... */
    }
```

Though it may seem like obvious guidance, if you are getting the error despite having called `load` and `sync` already, make sure that you're calling the `load` statement *on the same object that you're reading the property from*. That is,

- Don't re-fetch the object anew (see "Common mistake: re-invoking `get` methods" above.
- Don't call the load on a different object type from the object you're using (i.e., if you want to access `range.values` but instead call `worksheet.load("values")` – an actual real-life case that I've witnessed – can you really blame Office.js for claiming that `range.values` hasn't been loaded?..)

7.4.5 A rarer and more befuddling case: when using an object across `run`-s

There is one more case where you may encounter the `PropertyNotLoaded` error, despite having seemingly-corrected code. This case involves trying to re-use an object outside the linear flow of `Excel.run` (`Word.run`, etc.). Please see "*A common, and infuriatingly silent, mistake: queueing up actions on the wrong request context*" for more information.

7.5 Methods that return "primitive" types (strings, numbers, etc). E.g.: tables.getCount(), chart.getImage(), etc.

The preceding chapter, and even an earlier section on proxy objects all described how to load properties on regular API objects. There is one special class of objects that do *not* need a `load`, which get loaded automatically as part of a `sync`.

This type of object is a `ClientResult`[4], and it contains exactly one property: `value`. You can think of it as a wrapper over an actual primitive value (string, number, etc.). The only reason that a developer would ever fetch one of those objects is to read back the value… and so we may as well just pre-fetch it.

When and how is a `ClientResult` returned? A ClientResult is typically the result of a method call, where a *logical* type would be a primitive of some sorts (generally a string). For example, on a Word `Paragraph` object, there are `getHtml()` and `getOoxml()` methods[5]. Likewise, on the Excel `Chart` object, there is a `getImage` function that takes in the desired with, height, and fitting mode, and returns a base-64 representation of the chart image.

For all three methods, you might well expect their return type to be a string. That expectation is close to true – but if you look at their IntelliSense, you will see them listed as returning `ClientResult<string>`[6]. In other words, it's not a string object, but rather an object that wraps a `value` property that *is* the string.

What is the purpose of this strange misdirection? It comes down to a technical limitation, which is best explained with a concrete example.

[4] For Excel API developers out there: don't confuse `OfficeExtension.ClientResult` with `Excel.FunctionResult`. The latter is just a regular API object, no different than `Table` or `Font`: It contains more the one property (both `value` and `error`), and it can be used as a parameter to a method call to chain formula values. In short, for `FunctionResult`, you will need to load it explicitly just like you would any other proxy object.

[5] `Ooxml` stands for "Office Open XML" – the internal XML format of the document.

[6] For those unfamiliar with this notation: the angle-brackets are TypeScript's syntax for generic types. In this particular case, it denotes that the returned object is of type `ClientResult`, whose inner type – which in this case is the `value` property – is a `string`.

Suppose you have a chart proxy object. As a proxy object, it tries to create the *illusion* of local methods and properties, but ultimately it's only the *host application* – during a `context.sync` – that can load the correct base64 string. This means that if `getImage` were to return a primitive `string` object directly, there would be no way for `context.sync` to later re-substitute the actual value into the variable[7]! Instead, `getImage` must return a *boxed* type – which we call `ClientResult` – so that after a `context.sync`, Office.js can update the result's `value` property. And that way, you would use the `ClientResult` as follows:

```
Excel.run(async (context) => {
    // ...

    var image = chart.getImage(500, 300);
    await context.sync();

    $('#rendering').attr('src',
        "data:image/png;base64," + image.value);

}).catch(...);
```

You may wonder, why have `ClientResult`-s at all, if regular properties were good enough for nearly all the other properties in the OM (and are more convenient to use, since you don't need to create a separate variable for them). There are two reasons for why a value would have been represented as a `ClientResult` rather than a regular property.

1. The value is conditional based on certain parameters (in the example above, the arguments passed in as arguments to `.getImage`). This makes it a natural candidate for a method, as there isn't anything else it could be!

[7] In case this wasn't altogether clear: imagine you have `var a = 5`. You can now have `var b = a`, in which case `b` will also report as having value 5; and you can also set `b = 10` to switch `b`-s value. But there is nothing you can do with `b` to change the fact that `a` is still 5.

On the other hand, imagine that you now have `var a = { value: 5 }`, and you set `var b = a`. Naturally, if you now set `b` to a different object, such as `b = { value: 10 }`, the value of `a` will remain *un*changed. **But**, if you instead keep `var b = a`, and set a property on `b` – for example, `b.value = 10` – the change of the `value` property will be reflected in `a` as well, since they're still pointing at the same outer object.

2. The value is either very large (e.g., an OpenXML string for an inlined image in a paragraph), or expensive to compute (e.g., HTML, which get rendered on-the-fly during the API call). As you may recall from "The Golden Rule of `object.load`", if a developer forgets to specify property names in an `object.load()` call, all scalar properties on the object get loaded. This means that if such large or computationally-expensive values were to be represented as properties on the Paragraph object, a developers who only wants to read back the `text` property might inadvertently load a megabytes' worth of OOXML!

In terms of guidance: if you see a function whose return type is `ClientResult<X>`, just store the result value into a variable and call `context.sync` somewhere downstream, before accessing the variable's `.value` property – just like in the example above.

8. More core `sync` concepts

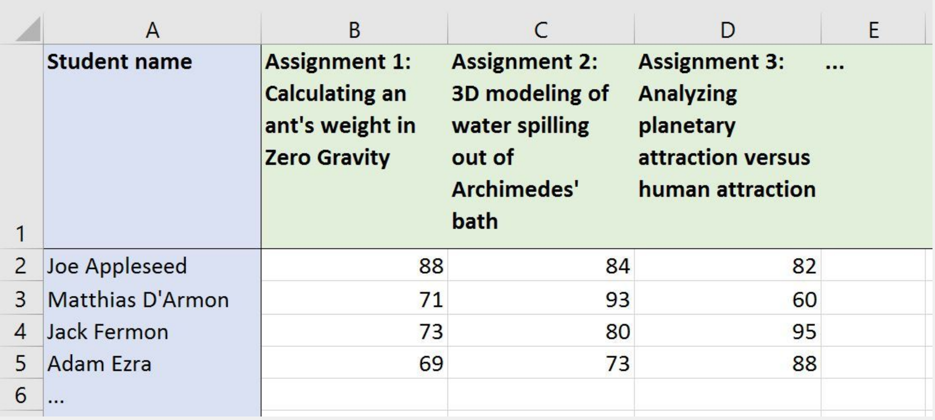
Chapter structure

8.1 Real-world example of multiple `sync`-calls

Enough with theory! Let's try out a real-world multi-`context.sync()` using a simple but reasonably-realistic example.

SCENARIO: A grade book tool

Imagine you are a math teacher, whose grade book is stored in Excel. Each row represents a student, and each column represents an assignment.



	A	B	C	D	E
1	**Student name**	**Assignment 1: Calculating an ant's weight in Zero Gravity**	**Assignment 2: 3D modeling of water spilling out of Archimedes' bath**	**Assignment 3: Analyzing planetary attraction versus human attraction**	**...**
2	Joe Appleseed	88	84	82	
3	Matthias D'Armon	71	93	60	
4	Jack Fermon	73	80	95	
5	Adam Ezra	69	73	88	
6	...				

You are preparing for a parent-teacher conference. When you meet with each parent, **you want to be able to show them *just* the data for their child, and filter the view to just assignments where the student got a *less-than-80% grade*** (so that you can discuss areas for improvement). Let's assume you'll be doing all the filtering in-place: you're not exporting the data to new sheets or anything like that, it's purely something that will be used for discussion with one set of parents, before being re-filtered for the next set of parents.

To give a concrete example: for the three assignments in the image above (and let's assume there's a whole bunch more, omitted for brevity), when talking with Matthias D'Armon's parents, the teacher would only want to show columns A (to see the student's name) and columns B & D from the assignments category. Column C, meanwhile, does *not* need to be shown, as it's the one assignment were Matthias got an over-80% grade (*might I say he had a "Eureka!" moment?*) – and so this isn't a problem area that the teacher needs to discuss with Matthias' parents.

This scenario is reasonably similar to the other two scenarios mentioned earlier – but with the notable exception that here you only want to look at *one student's* data, which represents a small fraction of the data on the sheet. I would argue that transferring a whole bunch of unneeded data is even worse than doing an extra sync[1], so let's see how we can do this task most efficiently, even if it means fudging a bit on the minimal-syncs principle for the sake of the avoiding-reading-copious-amounts-of-unneeded-data principle.

I think the most efficient breakdown of tasks is as follows:

STEP 1: Use the left column to find the student name.

First off, we need to find the student, so we can load just his/her data. This could be done by creating a Range object corresponding to `A:A`, trimming it down to just its used range (to reduce the one-million-plus cells into something that makes sense), and loading the cell values. However, in this particular case, we have even more efficient means: we can use an Excel function – invoked from JS – to do the lookup for us. The Excel function that fits this particular scenario is **`=MATCH(...)`**[2], which looks up a particular value and returns its row number[^row-number-1-indexed]. We'll assign the function result to a variable and load its "value" property.

Side-note regarding invoking `context.workbook.functions.match(...)`:

[1] My rationale – and of course, it really depends on the amount of extra data, but let's imagine that it's substantial, like 20x or 100x of what data will *actually* get used – is that both an extra network round-trip and extra bytes over the wire result in extra processing time. However, extra data might *also* cost the user extra money (on a cellular network, for example, where there are often substantial bandwidth caps), and will also put unneeded strain on the computer or server that is hosting the document (i.e., temporarily making the UI less responsive, or counting against the CPU/memory caps of Office Online). So, in my opinion, an extra sync is the lesser of two evils.

[2] **MATCH(lookup_value, lookup_array, match_type)**: Returns the relative position of an item in an array that matches a specified value in a specified order. *For more information, see https://exceljet.net/excel-functions/excel-match-function, or one of many other online resources for Excel formulas.*

> As noted in the intro chapter, the one exception to the Office.js 0-based indexing is when interacting with *numbers that are seen by the end-user*, or *parameters or function results of Excel formulas*. Hence, when we receive the result of `match(...)`, it will be 1-indexed, so we'll need to subtract 1 in order to interface correctly with the rest of the APIs.

Since we can't proceed with any further operations without knowing the row number (and it, in turn, can't be read without first doing a `sync`), perform a **sync**.

STEP 2: Retrieve the appropriate row, and request to load values

From the row number, retrieve the appropriate row, trim it down to just the used range (there is no sense in loading values for all 16,384 columns, when likely it's only a few dozen that are used), and load the cell values.

Again, without reading back the values, there is nothing further we can do. So, **sync**.

STEP 3: Do the processing & document-manipulation

Having retrieved the values, hide any column where the score is equal or greater to 80%, since the teacher's goal is only to discuss problem areas in this meeting (we'll assume that complementing *good* grades will have been done separately). This is where we finally issue a bunch of *write* calls to the object model, where we're manipulating the document, instead of just reading from it. So, to *commit* the pending queue of changes, do a final **sync**.

This seems like a reasonable plan, so let's code it up. **I encourage you to try this out yourself as an exercise**, before flipping to the following page and seeing the finished code.

Looking for a similar JavaScript-based (non-`async/await`) sample, instead?

For folks using plain ES5 JavaScript instead of TypeScript, you will find JS-only code for this identical scenario in one of the Appendix sections: "*JavaScript example of multi-`sync` calls*".

A three-'sync' automation task, for filtering the cells to a particular student and his/her less-than-stellar grades

```
Excel.run(async (context) => {
    // #1: Find the row that matches the name of the student:

    let sheet = context.workbook.worksheets.getActiveWorksheet();
    let nameColumn = sheet.getRange("A:A");

    let studentName = $("#student-name").val();
    let matchingRowNum = context.workbook.functions.match(
        studentName, nameColumn, 0 /*exact match*/);
    matchingRowNum.load("value");

    await context.sync()


    // #2: Load the cell values (filtered to just the
    //     used range, to minimize data-transfer)

    let studentRow = sheet.getCell(matchingRowNum.value - 1, 0)
        .getEntireRow().getUsedRange().load("values");

    await context.sync();

    // Hide all cells except the header ones and the student row
    let cellB2AndOnward = sheet.getUsedRange()
        .getOffsetRange(1, 1).getResizedRange(-1, -1);
    cellB2AndOnward.rowHidden = true
    cellB2AndOnward.columnHidden = true;
    studentRow.rowHidden = false;

    // Turn the visibility back on for columns with low grades
    for (let c = 1; c < studentRow.values[0].length; c++) {
        if (studentRow.values[0][c] < 80) {
            studentRow.getColumn(c).columnHidden = false;
        }
    }

    studentRow.getCell(0, 0).select();

    await context.sync();

}).catch(OfficeHelpers.Utilities.log);
```

Download sample code

To view or download this code, follow this link:
http://www.buildingofficeaddins.com/samples/gradebook

8.2 When to sync

There are only a few valid reasons for doing a `context.sync`:

1. You are ***done* with the batch function**, having read and/or manipulated the document in whichever way the scenario saw fit. And so, this final `context.sync` is your declaration of *"Dear Excel / Word /etc: I'm done working with you for the time being. Feel free to resolve the* `.run` *Promise, and to clean up*[3] *any resources that you so kindly provided to me. I'll get back to you when I need you again."*
2. You are ***in the middle* of a batch function**, and you **can't proceed without reading back some data from the document**, which you've previously requested to be loaded. Because you're dependant on the loaded data, and the load can't be fulfilled without a `sync`, you need to sync.
3. You have a collection object (e.g., a Word `RangeCollection` object, retrieved by calling `range.search(...)`), and you want to perform an operation (i.e., call a function like `range.getTextRange(...)`) on each item. In order for the collection items to be populated and be accessible via the `.items` property, you *do* need to call the `load` method and then do a `sync`. This scenario of wanting to populate items, without needing to read back any properties, does in fact happen in real-life[4]. My recommendation is to just choose a single reasonable property (generally, `id`, `text`, or `name`, whichever is applicable), call `load` with this property, and then do a `sync` at some point before accessing the items. (Definitely *don't* do `.load()` with no properties specified, however, as that would load *all* scalar properties, and would hence be much more wasteful.)
4. You were holding on to some object, but then needed to go off and do some [possibly-lengthy] web calls, or have been waiting on some user input. Depending on the scenario, and on how paranoid you are, it may be worth to re-load and re-sync the data on the object in case it's changed in the meantime.

[3] See "*What does* `.run` *do, and why do I need it?*" for details on what gets cleaned up.

[4] In fact, it happens in an example in this book – see "*Re-hydrating an existing Request Context*" for a real-life example.

5. You are *in the middle* of a batch function, where you had requested some really-temporal object (something like selection or active worksheet). You want to get a reference to this fleeting object as soon as possible (before the user changes his selection), so even though you don't need any data from the document, you do a `sync` to ensure that the identity of the fleeting object is correctly captured.
6. *[A corner-case]*: You are *in the middle* of a batch function, and you've queued up some "write" operations… but then you need to go and fetch some information from the internet. You're not awaiting on anything from the document, but you feel that it would be kinder to your users if you show them *what work you have done*, rather than having it all queued up but un-dispatched [5].

THE GOLDEN RULE OF `CONTEXT.SYNC()`: The only real reason to do a `context.sync()` is either to signal completion of interacting with the Office application's APIs, ***or*** to wait on the result of an `object.load(...)`.

Combined with "The Golden Rule of `object.load`" (see preceding section), this means that **the only reason to see `context.sync()` *in the middle* of a series of API calls is if you actively need some data from the document before your current operation can proceed.** Remember that each `sync()` takes extra processing time, and so your job is to minimize them whenever your can.

In short, **only `load` when you to**, and **only `sync` when you can't go on** without performing the queued-up `load` (and additionally at the very end of the function). Any other use of "sync" is almost certainly unnecessary, and detrimental to performance.

[5] In practice, much as I don't like to leave the user hanging, I don't like presenting him/her with a half-baked automation job, either. So in the vast majority of the cases, I would **not** do a `sync` here – and in fact, I would ideally arrange my processing in such a way that the web request happens first or last, without breaking up my OM calls (just for aesthetics' sake).

8.3 The final `context.sync` (in a multi-`sync` scenario)

As mentioned earlier, the batch function passed in to `.run` **must** return a Promise (which, 99% of the time, would presumably be a `context.sync()`). If you're coding using *plain* JavaScript (not TypeScript), this means that if you're doing is dispatching commands to Excel/Word/etc *in just a single sequential block*, you absolutely must have a `return context.sync()` statement[6]. In the case of TypeScript, as long as you declared the batch function to be **`async`** (i.e., `Excel.run(async (context) => { ... })`), the `async` keyword will actually make your function return a Promise anyway, even if it's the equivalent of `return Promise.resolve()`.

Now, if you have *multiple* `sync`-functions, or if you are using TypeScript, you have flexibility with the last `sync`: namely, technically speaking, you *could* omit it (line #**6** in the TypeScript snippet below, or #7 in the *JavaScript* snippet below that one).

TypeScript version of a 'sync' that can theoretically be omitted

```
1       Excel.run(async (context) => {
2           // ... {retrieve and load an object}
3           await context.sync();
4
5           // ... {manipulate the object}
6           await context.sync()
7       })
8       .catch(...);
```

[6] Or at least, you must be returning *some* Promise, where `context.sync()` is the most natural one. Technically speaking, you could return a Promise via some other means, such as having a `return Promise.resolve()` (but in that case, why *not* return `context.sync()` for better clarity?) or returning a Promise that was obtained through doing non-OM work, such as a web call (but in that case, why not do the web call *before* the `.run` altogether)?

JavaScript version of a ‘sync‘ that can theoretically be omitted

```
    Excel.run(function(context) {
        // ... {retrieve and load an object}
        return context.sync()
            .then(function() {
                // ... {manipulate the object}
            })
            .then(context.sync)
    })
    .catch(...);
```

Just because you *could* omit it doesn't mean that you *should* omit it, though – and I'll provide some reasons momentarily, for why I think including the `context.sync` should be a best-practice. However, in case you end up writing something like this by accident, and are bamboozled by how a `sync` is seemingly unnecessary, be aware that:

- Yes, this is in fact real behavior; you're not imagining things.
- This only works for *the last* `sync`*-function*. So, don't take this as *carte blache* to skip over other `sync`-functions in the middle of your batch!
- The automatic `sync` only works within `Excel.run`, `Word.run`, etc. – but *not* in case you're using a "*raw*" request context outside of a `.run` (see "*When to use a raw RequestContext in place of an Excel.run*").

Now, for why this works at all: before beginning the cleanup process at the completion of the batch, the `.run` method will check if there are any pending actions, and do an *automatic* `sync` if there is anything left in the queue. This was done for developer convenience: it seemed inevitable that some of the time, people would forget to do the last `sync`, causing their last chunk of code to effectively no-op in a silent and unexpected way. As I am philosophically against unexpected no-op-s (and while I would have been involved in the design discussions anyway, in this case *I was the one implementing* `.run`), we had two remaining choices: either *throw an error* or *silently do the right thing*. From a teaching perspective, throwing an error would have been more consistent, but it felt overly harsh, especially on beginners (who were more likely to forget to add proper error-handling in any case). On the other hand, it was obvious to guess the developer's intent here; and it aligned nicely with a quirky phrase from my previous team, of ensuring that if a developer makes an error, he/she still *falls into the pit of success*. And so, we opted to do an automatic syncing behavior if the last `sync` in a multi-`sync` batch is omitted.

That being said, I still think that it's valuable to include the final `context.sync` for the following reasons:

1. It reinforces the pattern of

   synchronous-om-call ⇒ **sync**;
   synchronous-om-call ⇒ **sync**;
   synchronous-om-call ⇒ **sync**.

2. It ensures that errors in the last portion of your batch will get bubbled up correctly (they still will if you put a `.catch` outside of the `.run`, but in practice I've seen a lot of developers put their `.catch` inside the batch function).

3. If you or a colleague later decide to append more code, you will already have the `context.sync` in place, thereby ensuring that you don't forget to add it when the need comes.

4. Finally, it adheres to the principle of least surprises, by being explicit about a `sync` that would happen under the covers anyway.

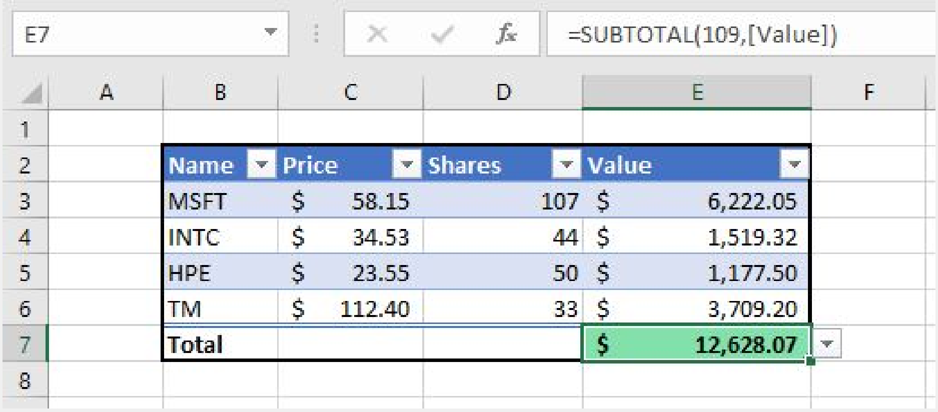
Whatever you do, the *worst* thing you can do is include a `context.sync()` invocation but *forget* to `return` or `await` it. This would lead to a broken Promise chain and all the evils therein. So, both with the last `context.sync` any any other, remember that `sync` statements should *always* be awaited/returned.

8.4 A more complex `context.sync` example:

If the code sample from a few pages ago – “*Real-world example of multiple* `sync`*-functions*” – made sense, lets try something with even more `sync`-functions, and that feels even more like a “real-world” scenario. [Conversely, if the previous code sample did *not* make sense, it may be worth looking over it again before proceeding with this one].

Scenario: Stock-tracker and net-worth calculator

Imagine you have an Excel table, where you keep track of stocks that you own. You keep multiple copies of the table, one in each sheet, with each sheet corresponding to a particular year (or month, or week… basically, snapshots in time). The last (rightmost) sheet is your *current year’s* sheet – which you want to update based on the current stock values fetched off of the internet. You also want to read the previous sheet’s total-stock-worth and compare it with the latest sheet’s total. Depending on whether the latest value is better or worse than the previous sheet’s value, you will color the cell representing the total in either green or red. (For the stock-fetching function, let’s use `getStockData` from *the last example* on section “*Creating a new Promise*”).



E7

=SUBTOTAL(109,[Value])

	A	B	C	D	E	F
1						
2		Name	Price	Shares	Value	
3		MSFT	$ 58.15	107	$ 6,222.05	
4		INTC	$ 34.53	44	$ 1,519.32	
5		HPE	$ 23.55	50	$ 1,177.50	
6		TM	$ 112.40	33	$ 3,709.20	
7		Total			$ 12,628.07	
8						

As in the previous example, before writing the code, let's do a quick outline of the steps (and `sync`-s) involved.

1. First, load the worksheet collection in order to obtain – after a sync – references to the last two worksheets. Note that since you can't load a collection with no properties on the children, you have to load at least one property. Let's load the `name` property on the sheets, since it is both short (won't transfer too much data over the wire), and useful for displaying messages in the UI(e.g., *"Data on sheet 'Stocks 2016' successfully refreshed"*). So: load and **sync**.
2. On the "latest" worksheet, get the first table on the worksheet (we'll assume that there is only one table per sheet – though we could validate, if we really wanted to). Load the values of the leftmost column of the body of the table (the stock names). And again, **sync**.
3. Using the stock names obtained in the previous step, format and issue a web query to obtain the latest price data. This is an asynchronous operation, but not an Office-involved one. So *no sync*.
4. Because the stocks web-server call may have taken a while – and depending on how paranoid we are – it may be best to re-load the table values, in case the user had rearranged or deleted stock names in the process[7]. Having re-issued the call for re-loading the stock names, **sync**.
5. For each stock symbol, add the appropriate values to the price column. With the value-setting dispatched, queue up the loading of both this table's "Total", and the "Total" from the previous worksheet. (We are reading back the data, instead of computing the current total, so that if the "Total" involves formulas or some other Excel wizardry, that read it back correctly, just as it would show on the worksheet). **sync**.

[7] Realistically, both in the case of Excel multi-user co-authoring, and in the add-in running while the user is taking actions, it should be noted that Excel is *not* designed for "battleship Excel", as one developer on the Excel team called it. If there are two different users, both editing the same area in the document, you may end up with conflicts regardless whether or not there are Add-ins thrown in the mix or not. Concrete example: the user looks up a particular stock symbol for a cell, goes into the browser to find out the current value, and enters it in to the appropriate cell. However, at that very moment, a co-author had moved the stock symbol on that row down by one cell, and had written a different symbol in this row's place. Both users may have a temporarily-consistent view of their workbook, but an erroneous mismatch between stock symbol and value after the two co-authoring states are merged into the "eventual consistency" that's reflected in the document.

6. Having compared the current and previous totals, assign a green (positive) or red (negative) background to the current total cell, and issue one final **sync** command to commit this action.

Splitting work across multiple subroutines

In this example, I will split the code into multiple subroutines – both for readability in book form, and also for showing my preferred approach to splitting work into functions in Office.js

The approach consists of one main function, which is essentially connected to a button click or some other user interaction. This overarching function would perhaps show some "loading" UI, and then let the `Excel.run` execute to completion, ensuring that it has appropriate error-handler built in. The function might also, at the end, do some UI post-processing, like removing the "loading" overlay or displaying a success notification to the user. For brevity's sake, I'll only show the `Excel.run` portion here, since that would stay the same no matter the UX paradigm; by the same token, I'm using simple console-logging for my error-handling, but any real-world application should have user-facing error UI.

Within this main subroutine, I will sometimes do OM (object-model) calls directly, if it's only a line or two; otherwise, I will call the helper subroutines. You can tell which ones are helpers by the fact that they receive an API object as an incoming parameter; whereas the master subroutine would not, as it would establish its own `Excel.run`. The master subroutine would also add error handling, so the helpers don't need to worry about it, and can be as lean as can be.

The idea is that whether my OM call is inlined or separated into a helper function, *all it does is set up the appropriate proxy objects*, but it never calls `context.sync()` – only the "master" is allowed to do that. The benefit of this approach is that the asynchrony is easy to manage (it's in one place), and that there are no wasted `sync`-s (as I can call multiple helper functions or do other quick OM calls before invoking the `sync`). To each such helper function, I pass in only the objects that it needs (e.g., a previously-fetched worksheet or range).

OK, enough planning. Let's write some code! (Again, I recommend that you try this yourself before looking at the finished code on the next pages).

The main ‘Excel.run‘ function of the multi-sync, stock-fetching scenario

```
Excel.run(async (context) => {
    console.log("#1: Obtain and loading worksheets");
    let sheets = context.workbook.worksheets.load("name");
    await context.sync();

    console.log("#2: Retrieve stock table & values. " +
        "Also load tables on previous sheet, if any");
    let latestSheet = sheets.items[sheets.items.length - 1];
    let latestTable = latestSheet.tables.getItemAt(0);
    let stocksRange = latestTable.getDataBodyRange()
        .getColumn(0).load("values");
    let previousTables = loadPreviousSheetTables(sheets);
    await context.sync();

    console.log("#3: Issue web request to read stock data");
    let stockNames = stocksRange.values.map(rowData => rowData[0]);

    // With this names-list in hand, call the "getStockData"
    // function from an earlier chapter ("Promises Primer"").
    // The function returns a Promise, so await it:
    let stockData = await getStockData(stockNames);
    console.log("Received stock data", stockData);

    console.log("#4: Reload stock names, to ensure no " +
        "changes occurred while waiting on the web request");
    stocksRange.load("values");
    await context.sync();

    console.log("#5: Write in the updated stock prices, " +
        "and load the new total (and previous, if any)");
    updateTable(latestTable, stocksRange, stockData);
    let latestTotalCell = latestTable
        .getTotalRowRange().getLastCell().load("values");
    let previousTotalCellIfAny =
        loadPreviousTotalCellIfAny(previousTables);
    await context.sync();

    console.log("#6: If there was a previous 'Total' to " +
        "compare against, color the current one accordingly.");
    addCellHighlightingIfAny(
        latestTotalCell, previousTotalCellIfAny);
    await context.sync();

    console.log(`Done! Data on sheet "${latestSheet.name}" ` +
        "has been refreshed");

}).catch(OfficeHelpers.Utilities.log);
```

And here come the helper functions, which either return an API object, or just operate on the passed-in parameters:

The fully-synchronous-OM helpers that comprise the larger multi-sync scenario

```
function loadPreviousSheetTables(
    sheets: Excel.WorksheetCollection
) {
    let sheetCount = sheets.items.length;
    if (sheetCount >= 2) {
        return sheets.items[sheetCount - 2].tables.load("name");
    }
    return null;
}

function updateTable(table: Excel.Table,
    stocksRange: Excel.Range, data: any
) {
    let pricesToWrite = stocksRange.values.map(row => {
        let stockName = row[0];
        let priceOrEmptyString = data[stockName];
        if (typeof priceOrEmptyString === "undefined") {
            priceOrEmptyString = "";
        }
        return [priceOrEmptyString];
    });

    let priceColumn = table.columns.getItem("Price")
        .getRange().getIntersection(stocksRange.getEntireRow());
    priceColumn.values = pricesToWrite;
}

function loadPreviousTotalCellIfAny(
    tables: Excel.TableCollection
) {
    if (tables && tables.items.length === 1) {
        return tables.items[0].getTotalRowRange()
            .getLastCell().load("values");
    }
    return null;
}

function addCellHighlightingIfAny(
    latestCell: Excel.Range, previousCell: Excel.Range
) {
    if (previousCell) {
        let isLatestGreater =
            latestCell.values[0][0] >
            previousCell.values[0][0];

        latestCell.format.fill.color =
            (isLatestGreater ? "#82E0AA" : "#EC7063");
    } else {
        console.log("Skipped comparison with previous " +
            "total, as there doesn't appear to be one");
    }
}
```

By the way, note that the last helper, `addCellHighlightingIfAny`, will sometime add an OM operation to the queue, and sometimes not. In the latter case, the `await context.sync()` in the master function will end up with no pending actions – and that's OK! The `sync` is smart enough to short-circuit and no-op rather than making an empty round-trip to the host, so that's not a problem at all.

Download sample code

To view or download this code, follow this link:
http://www.buildingofficeaddins.com/samples/stocks

8.5 Avoiding leaving the document in a "dirty" state

As mentioned in earlier sections, there is no *atomicity* to the `Excel.run` or the `sync`: any actions that have already been done will *stay* done. For example, suppose you have the following code (repeated from the *Implementation details* section):

```
1     Excel.run(async (context) => {
2         let range = context.workbook.getSelectedRange();
3         range.clear();
4         let thirdRow = range.getRow(2);
5         thirdRow.format.fill.color = "purple";
6 
7         await context.sync();
8     }).catch(OfficeHelpers.Utilities.log);
```

During the processing of the queued-up requests, suppose that the action to retrieve the third row fails (e.g., the selection was a 2x2-sized range). In this case, the *clearing* of the selection will already have been done, and so when the execution aborts, the selection will remain cleared (even though no third row ever got set to purple). While not ideal, this is no different from VBA or VSTO with regards to Office automation; it is simply too difficult to roll back, especially given any user or collaborator actions that may have happened in the meantime. Excel is a user application with loose structure – it is *not* a database or a bank ATM.

To this end, my recommendations for error-handling are as follows:

1. Pre-validate anything you can, before performing any document-impacting actions. That is, pre-fetch all objects that you need, load and check any properties, and only then begin the actual document-manipulation portion. In case of the example above, a more optimal flow would have been to swap lines #3 and #4, so that if the third row isn't found, the whole chain of document-manipulating commands never starts.
2. In a similar vein, pre-validate any user input. For example, if the fill color ("purple") came from a user-defined field, it would be best to validate that it's one of the valid HTML color names.

3. In case of an error, inform the user via a detailed message, ideally explaining what the function was supposed to do, so that the user can reason about what he/she should *undo* (e.g., delete the incomplete sheet that was being compiled when the function failed). If it would help the user to know where exactly the operation failed, it's possible to narrow down the exact location; see "*Finding the point of failure: Trace Messages*". The *very* good application might even intercept the error, inform the user, and ask if it should attempt to undo the damage (i.e., "*would you like to have the incomplete report sheet deleted?*"). Depending on the host application and platform, the user might be able to click "undo" to recover, but don't take that for granted; and in the case of Excel, be aware that the undo stack gets blown away as soon as a document-manipulating action occurs (same as VBA/VSTO). So by "*undo*" in the preceding sentences, what I mean is a *contextual* undo done by the user or program based on their knowledge of what actions had taken place; not the application's `ctrl+z` sort of undo.

9. Checking if an object exists

Suppose you want to operate on an object (say, a worksheet named “Report”), but you first need to check if the item exists. Or maybe you want to get an intersection of the user’s current selection, and data in a particular table column, and check that there is in fact an intersection (i.e., the two ranges aren’t completely disjoint). How would you do that?

There are two ways of accomplishing this task.

Chapter structure

- **Checking via exception-handling – a somewhat heavy-handed approach**
- **A gentler check: the `*OrNullObject` methods & properties

9.1 Checking via exception-handling – a somewhat heavy-handed approach

On collections – for example, the Worksheet collection in Excel – you will often see a `.getItem(key)` method that takes in a string parameter (an ID or name). Likewise, you will see other `get`-prefixed functions (such as the Excel's `range.getIntersection(otherRange)`. If you use these functions as is, and if the object fails to exist, the `context.sync()` will immediately abort its operations and exit, throwing an exception back to the JavaScript, in the form of a rejected promise. Provided that you have an error handler, the error will get bubbled up. And so, if you intercept it and do so intentionally, you absolutely can use this for checking for an item's existence or validity.

The approach, however, leaves something to be desired. In particular:

- Because any operation following the unsuccessful item-fetching (whether via method or property) will throw, this effectively means that you need to check for one item at a time. If you have X things to check, you will need to have X `sync` calls to validate each and every one of them.
- Similarly, if you want to do an operation only if an object exists, and no-op or do some alternate action otherwise, you end up needing an extra `sync` and [intentional] `catch` before proceeding with your logic.

For the second issue, here's a concrete example. Suppose you want to delete a sheet if it exists, and then create a brand new one in its place (or just create the new one). In fact, if you've been using Script Lab, you will see that many Excel samples use a method from OfficeJsHelpers for this very scenario, called `ExcelUtilities.forceCreateSheet`. Let's derive our own version of it, if all we had to rely on was just `catch`-ing errors:

Taking actions based on object existence, using only a ‘catch‘ approach

```
async function createReport(sheetName: string) {
    // ... {some preceding code}

    // Flush anything already in the queue, so as to
    // scope the error handling logic below.
    await context.sync();

    try {
        let oldSheet = workbook.worksheets.getItem(sheetName);
        oldSheet.delete();
        await context.sync();
    }
    catch (error) {
        let isExpected = (error instanceof OfficeExtension.Error &&
            error.code === Excel.ErrorCodes.itemNotFound);

        if (isExpected) {
            // This is an expected case where the sheet
            // didn't exist. Do nothing
        } else {
            // Re-throw the error, as it must have been something
            // external that caused it, not the missing sheet
            throw error;
        }
    }

    context.workbook.worksheets.add(sheetName);

    // ... {do something more with the sheet}

    await context.sync();
}
```

The code above, in slightly different form, is actually still used by `OfficeJsHelpers` as a fallback for older clients (though the `OfficeJsHelpers` one has another safety check, which I omitted here for brevity[1]).

In short, while it’s possible to make use of the error-throwing behavior for detecting missing objects, a throwing behavior is both less convenient, and just feels a wee bit extreme.

An analogy

For variety’s sake, let’s take an example outside the realm of programming:

Let’s say you approach a llama at a petting zoo, or a camel in the desert (happens all the time, right?). *“Hey cute little buddy, can I pet you?”*. Three possibilities:

- Llama comes to you and nuzzles against your hand. Answer is “yes”.
- Llama looks at you, then shies and backs away: Answer is “no”.
- Llama spits at you. The answer is still “no”, but don’t you think it could have been expressed in a gentler way?

[1] What makes my code less safe is that I’m making an assumption that there will be another visible sheet in the workbook when this function is called. Excel requires that you always have at least one visible sheet, so if the existing sheet was the one and only visible one, a failure would be thrown during the attempt to delete call (and re-thrown by the catch). So the safer version, in the generic case, is to first create the sheet (that way you’re guaranteed to have one), and only then attempt to do the deletion; and finally, after the deletion or lack thereof, rename the sheet to what you actually wanted it to be called. This is what I meant earlier in the book, when I said that you should be an expert in the product that you’re automating… because unexpected quirks like these are unavoidable, so you should be prepared to repeat your automation steps manually, seeing what errors the UI throws at you (which will often be more detailed than the API-provided ones).

9.2 A gentler check: the `*OrNullObject` methods & properties

Because throwing an exception is not always optimal, many methods offer an `OrNullObject` variant, which is non-throwing. The concept is a combination of the "null-object" design pattern[2], and the null-propagation mechanism found in C# (starting with 6.0), where you can make a series of calls using the `?.` syntax, such as:

```
string title = section?.GetPage("XYZ")?.Title;
```

In JavaScript, there is no special syntax for null-propagation – and even if there were, it might still not have helped our Office.js scenario, given that the actual execution happens on the server, not in the browser. But, this is where the methods suffixed with `OrNullObject` come in. Using these methods, you can write code like this:

```
let reportSheet = context.workbook.worksheets.getItemOrNullObject("Report");
let summaryTable = reportSheet.tables.getItemOrNullObject("Summary");
let range = summaryTable.getDataBodyRange();
range.load("values");
```

... And notably, this code will execute in a *non-throwing fashion* even if the Report worksheet and/or the Summary table *don't exist*.

This should surely sound strange. In fact, it very much is: Null Objects are strange beasts, that defy normal conventions! In terms of taming and using such objects, the scenarios fall into two categories:

9.2.1 Case 1: Performing an action, and no-op-ing otherwise

If the reason that I was fetching the "Summary" table above was to delete it if it exists and no-op otherwise – quite a common scenario, when you want to ensure unique naming and re-export some data, I can just invoke the action on it directly: `table.delete()`. Likewise, if I was fetching the table as part of a

[2] See https://en.wikipedia.org/wiki/Null_Object_pattern

general formatting routine (but am OK with no-op-ing for elements that don't exist in the document), I can just call just perform the call as if it's a real object:

```
table.getRange().getColumn(0).format.font.bold = true
```

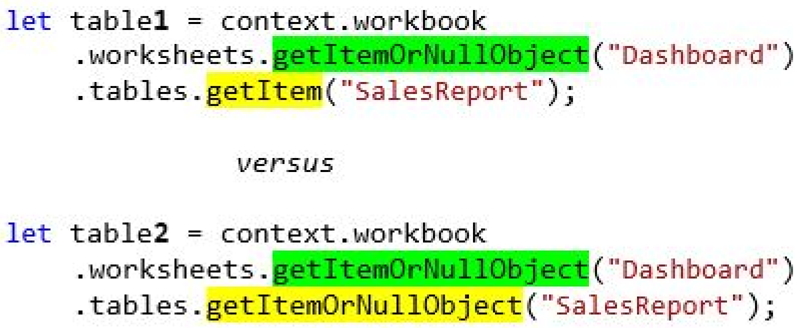
These calls will dutifully perform the requested action if the object (e.g., table) exists, and will silently do nothing if it doesn't. This no-op behavior was added specifically for chaining "get" commands (i.e., being able to fetch the first column of a particular table on a particular worksheet, in and only if the whole chain exists, without needing to do a bunch of null checks and `context.sync()`-s). But, as you see above, this no-op behavior extends to any other method calls or property value-setting, and can be particularly valuable for the "*do this if the object exists, and never-mind otherwise*" behaviors.

Important: Objects may be weirder than they appear, once you enter `*OrNullObject` territory!

An object derived from a `OrNullObject` method (or the child, grandchild, sibling, etc. thereof) **will look like a regular object, but it *may* have a completely different (namely, no-op-ing) behavior** for the cases when the object doesn't exist.

Moreover, the "null-object"-ness spreads to any object derived from that object: so once you've fetched a proxy object that corresponds to a missing worksheet, any subsequent calls to the tables collection, or to a range, or to anything else off of that object will result in other null objects. This behavior is what makes chaining "get" commands across null objects possible, but it will also get you into trouble if you forget that you're operating under null-object rules.

Note that when chaining object accessors, there is a subtle but important difference between the following two ways of accessing a table off of a null object:



```
let table1 = context.workbook
    .worksheets.getItemOrNullObject("Dashboard")
    .tables.getItem("SalesReport");

                versus

let table2 = context.workbook
    .worksheets.getItemOrNullObject("Dashboard")
    .tables.getItemOrNullObject("SalesReport");
```

Subtle difference in accessing an object derived from a null object

What is the difference in behavior? It depends on the condition of the workbook state:

- If both the “Dashboard” sheet AND the “SalesReport” table exist, then both `table1` and `table2` will amount to the same thing (and would have worked perfectly fine even with just a `getItem` for each).
- If the “Dashboard” sheet does ***NOT*** exist, both `table1` and `table2` will ***also*** have identical behavior (this time resulting in a never-throwing null object), as the null-ness of the missing “Dashboard” sheet gets propagated to the “SalesReport” table regardless of how it’s fetched.
- Finally, if the “Dashboard” sheet exists ***BUT “SalesReport” does NOT***, you get diverging behavior. In the case of `table1` (a `getItem` for a missing table), the missing item will cause a throwing behavior, *despite* being derived from a call that had a `getItemOrNullObject` in its ancestry! Why? Because, once the sheet is fetched and proves to be a real valid worksheet object, the fact that it came from an `OrNullObject` function no longer matters (hence the getItem**Or**NullObject, not a getItem*As*NullObject)… And so the rest of its behavior (namely, throwing when asked for a non-existent table) remains as is.

As you look at the above code, there is no doubt that the `OrNullObject` suffix stands out. Could the Office Programmability team not have come up with a shorter name? We undoubtedly could have – but truth be told, we wanted to make it *painfully and explicitly obvious* that you’re entering the “null-object” territory, and that, once there, things will go terribly awry if you’re not expecting that object will gleefully *do nothing* when asked.

This brings us to the second use cases for `*OrNullObject` methods, where instead of just no-op-ing, you actually want to check whether you're in null-object land or not.

This brings us to the second use cases for `*OrNullObject` methods, where instead of just no-op-ing, you might actually want to check whether you're in null-object land not.

9.2.2 Case 2: Checking whether the object exists, and changing behavior accordingly

In this second case, you actually *do* care about whether the object exists or not – and so the use of `*OrNullObject` is merely a convenient way of confirming an object's existence. The advantage of using `*OrNullObject` methods for this use case, over their throwing counterparts, is that you can simultaneously check many different objects as part of one batch, and you do not need multiple `context.sync()` calls for each level in the call hierarchy.

To find whether the object (whether derived directly from a "*OrNullObject" method, or a descendant from a chain of null-object calls) is indeed a null object, you would:

1. Perform the `*OrNullObject` method or property invocation, storing the resulting object (or its child, grandchild, etc) into a variable. For example:

```
let tableIfAny = sheet.tables.getItemOrNullObject("SalesReport");

let intersectionIfAny =
    sheet.getRange("C:C").getIntersectionOrNullObject(
        context.workbook.getSelectedRange());
```

2. Invoke `context.sync()`. This **need not** immediately follow the "*OrNullObject" method/property invocation – in fact, if you're batching multiple queries for several possibly-null-object objects, it certainly couldn't be! The only important part is that you perform the `context.sync()` at some point before trying to access the object.

3. Check the `isNullObject` property on the object (i.e., `if (tableIfAny.isNullObject) { ... }`). If IsNullObject returns true, it means the object didn't correspond to anything real in the document (e.g., there was no table named "SalesReport" in the workbook, or there is no intersection between two ranges, etc.). Based on this information, you can now perform whatever action you would have done if you knew this information (creating the missing worksheet, alerting the user, or whatever else).

A corollary to the `OrNullObject` pattern: there is _no need to check_ for `isNullObject` _unless_ you are in `*OrNullObject` land.

As mentioned earlier, objects derived from a `*OrNullObject` invocation will behave very differently from regular objects, in cases when the object is, in fact, missing. For this reason, methods or properties that return these "null objects" are explicitly sufficed with `OrNullObject`. _*As a corollary, any method or property that is *NOT_ suffixed with `OrNullObject` will throw an exception in case of an error. Thus, if the `context.sync()` succeeds for a regular method/property, you are guaranteed that the object did in fact exist (else it would have thrown), and so _there is never a need to check for the_ `IsNullObject` _property_ on regular methods & properties.

10. `Excel.run` (`Word.run`, etc.) advanced topics

Chapter structure & planned content

Note: Sections that have already been written, and are **included** in this book, are **bolded and hyperlinked**; the rest are planned topics, but have not been written (or polished) yet.

- **What does `.run` do, and why do I need it?**
- **Using objects outside the "linear" `Excel.run` or `Word.run` flow (e.g., in a button-click callback, in a `setInterval`, etc.)**
  - **Re-hydrating an existing Request Context: the overall pattern, proper error-handling, `object.track`, and cleanup of tracked objects**
  - **A common, and infuriatingly silent, mistake: queueing up actions on the wrong request context**
  - **Resuming with multiple objects**
  - **Why can't we have a single global request context, and be one happy family?**
- Awaiting user input
- Interrupting `Excel.run`
- When to use a raw RequestContext, in place of `Excel.run`

10.1 What does "`.run`" do, and why do I need it?

As part of `Excel.run`, in addition to creating a new request context, the method also does the following

1. Automatically manages the tracking & release of API objects

This is most important for the Range objects in Excel & Word, which have no unique ID and must therefore be specifically tracked [1]. Essentially, if it weren't for `Excel.run`, and if you were `new`-ing-up a Request Context object directly ("`var context = new Excel.RequestContext()`"), you would see a bunch of errors regarding "InvalidObjectPath" whenever you'd use a Range object across a `sync` (e.g., load and sync a Range's value, then set a property or perform an action based on the read data on that same object instance). While it's possible to use the `.track` and `.untrack` (e.g., `range.track()`) methods to control the object lifetime manually, it's both tedious and error-prone (and it's notoriously difficult to do proper cleanup for all the objects that were were created, esp. in error cases). The `Excel.run` (`Word.run`, etc.) methods free you of that responsibility, and do automatic tracking and untracking for anything that happens during the sequential flow of that particular batch. [Note that failing to release tracked objects will lead to the add-in consuming more and more of the host's resources, slowing down the application quite drastically once you have 100s or 1000s of such ”leaked” objects].

Note: You will still need to do `range.track()` if you intend to use that same object at some later point (e.g., when a user presses a button), outside of the linear "Excel.run" flow. But such cases become much less frequent, and is something that you'd generally do only for a few select objects, rather than being something you have to do all the time. See "*Using objects outside the "linear" Excel.run flow (e.g., in a button-click callback, in a setInterval, etc.*" for more information.

2. Automatically "flushes" the request queue

[1] If you're curious, see the section in the Implementation Details about *A special (but common) case: objects without IDs*.

The `Excel.run` (`Word.run`, etc.) methods automatically perform a final `context.sync()` at the end of the batch, even if the developer forgot to put it there. (Note that I still think that being explicit about your `context.sync()`-s is better, so I would treat this as more of a a safety net rather than something to rely on all the time; see section "*The final* `context.sync` *(in a multi-*`sync` *scenario)*".

3. Dispatches any pending "blocked" `Excel.run`-s at its completion

This one is a bit harder to explain; but if there are any pending `.run`-functions waiting to use the same request context, `Excel.run` will notify the first of them that it can now run. See "*Using objects outside the "linear" Excel.run flow (e.g., in a button-click callback, in a setInterval, etc.*", and this sub-section in particular, for more information.

4. Standardizes error-handing

Before the magic of `async-await` came to be, if you wanted to trap all errors in code that had a mix of synchronous and asynchronous code, you would need a standard `try/catch` block for the synchronous code, and then a separate `.catch(function(e) { ... })` handler for Promise failures. The `Excel.run` (`Word.run`, etc.) method standardizes all errors, ensuring that if a runtime JS error occurs before the first `context.sync()`, the exception is converted into a rejected Promise, just as it would if it occurred after the first `context.sync()`.

[Note that with TypeScript's 'async/await', this is no longer of particular concern. You can use a 'try/catch' block everywhere, for both synchronous and asynchronous code, so long as you're sure to 'await' all your Promises].

10.2 Using objects outside the "linear" `Excel.run` or `Word.run` flow (e.g., in a button-click callback, in a `setInterval`, etc.)

10.2.1 Re-hydrating an existing Request Context: the overall pattern, proper error-handling, `object.track`, and cleanup of tracked objects.

So far in this book, there was an implied assumption that the async nature of the APIs is a technologically-necessary evil, but that the APIs try their hardest to create an *illusion of synchrony* (even if punctuated by the occasional pesky `load` and `sync` statements). Thus, everything in the book so far treated scenarios as something that should execute in a linear (or seemingly linear) fashion. The very guidance to await your `context.sync()`-s and `Excel.run`-s stems from this assumption.

But what about the cases where you want *intentional asynchrony*? For example, imagine that you're implementing a search functionality in a Word document, where you want to list out some matching content in a taskpane, and then have the user be able to click on an item to perform some action on it (color it, select it, etc.). In such scenario, the user's click is going to come at some indeterminate time *after* you've already finished doing the initial `Word.run` that listed out the content. How do you resume using an object after letting it go?

Having observed users struggle with this when we first released the original Office 2016 APIs, we've added some convenience functions that you can use in ExcelApi 1.2+ and WordApi 1.2+ (essentially, anything later than the MSI-install of Office 2016[2]). Even so, there are some subtle nuances that you need to be aware of. So let's take it one step at a time.

First, let's begin with just the initial `Word.run` code that will populate the search list:

Try it out

[2] See the topic on Office versions.

If you want to follow along, just import the following Snippet ID into Script Lab:
dfb2693888d31062b636c65ac8c5259f

The initial ‘Word.run‘

```
$('#search').click(() => tryCatch(performSearch));

async function performSearch() {
    let searchTerm = <...>;

    await Word.run(async (context) => {
        let searchItems = context.document.body.search(
            searchTerm, {matchCase: false, matchWholeWord: true});
        searchItems.load("text");
        await context.sync();

        let fullSentences: Word.Range[] = [];
        searchItems.items.forEach(range => {
            let sentence = range.getTextRanges(
                [".", "?", "!", ",", ";"]).getFirst();
            sentence.load("text");
            fullSentences.push(sentence);
        });
        await context.sync();

        let $searchResults = $("#search-results").empty();
        fullSentences.forEach(range => {
            let $button = $("<a href='javascript:void(0)'>")
                .text(range.text)
                .click(() =>
                    tryCatch(() => selectAndFormatRange(range)));
            $searchResults.append($button);
        });
    });
}

```

```
function selectAndFormatRange(range: Word.Range) {
    console.log("TODO");
}

/** Default helper for invoking an action and handling errors. */
async function tryCatch(callback) {
    try {
        await callback();
    }
    catch (error) {
        OfficeHelpers.UI.notify(error);
        OfficeHelpers.Utilities.log(error);
    }
}
```

Now, onto the click handler. The trick is to do a Word.run just like you normally would, but instead of having it create a new anonymous request context, have the run **resume using the context of some existing object**.

To resume using an existing context, you simply use one of the function overloads available off of Word.run; namely, an overload that takes in an object (or array of objects) as the first argument, and the batch as the second:

Re-hydrating the 'Word.run' function with an existing request context

```
...
async function selectAndFormatRange(range: Word.Range) {
    await Word.run(range, async (context) => {
        range.font.highlightColor = "yellow";
        range.select();

        await context.sync();
    });
}
...
```

By the way, a small side-note: In both performSearch and selectAndFormatRange, I am, of course, await-ing Word.run (so as not to break the Promise chain), but I am ***not*** doing error-handling. This is because I structured the code in a way that performs the error-handling *at the invoker level*. I prefer this pattern, because it promotes code reuse and composition (and for a book rendered in fairly narrow width, it also avoid an extra level of indentation, which is always a nice bonus). In all seriousness, I do think that it's cleanest and most convenient to *only* do the error-handling at the topmost level, right where the user performs an action (e.g., when registering a button click handler). This is why my click handler takes in an anonymous function that calls tryCatch(performSearch), rather than taking in performSearch directly):

```
    $('#search').click(() => tryCatch(performSearch));
```

Why do I mention error handling? Because if you write out and run the above code – which I encourage you to do – you will notice that the code fails to execute. If you have proper error-handling, you will at least get a hint for *why* it is failing; namely, because of an InvalidObjectPath error[3].

Let's explore why. The code is actually very close to correct: if the Word or Excel Range objects had a persistent ID – just like Word content controls and tables, or Excel worksheets and tables and charts – the code would "just work". But as described in the Implementation Details section, with regards to "*A special (but common) case: objects without IDs*", objects without IDs need extra work to be kept in-memory. Inside of the linear flow of a Word.run, these objects would automatically get tracked as needed. But, once the Word.run Promise resolves, such objects automatically get cleaned up, to avoid leaking memory and slowing down the host application. The JavaScript proxy objects are still be available to the JavaScript runtime (following normal JS scoping rules), so this cleanup is different than true runtime garbage collection. But for all practical purposes, these objects becomes phantoms, a mere shadow of the objects' previously-glorious life. You can still read properties that have

[3] Actually, it looks like even though Excel correctly throws the "InvalidObjectPath" error for these cases, at the time of writing (July 2017), Word throws an "InvalidArgument" error instead. There is a bug tracking this, to make the error behavior be consistent and correct.

already been loaded, but invoking any methods or setting any properties on such object will result in an InvalidObjectPath error.

So: what do we need to do to fix this? It's actually quite simple: inside the Word.run code, in the places where it's still linearly executing, we need to declare our intention to make later use of these objects. To do this, we call the .track() method on the necessary objects. This will make the objects exempt from automatic cleanup.

The fix for 'InvalidObjectPath': calling 'object.track()' while still inside the linear flow of a 'Word.run'

```
async function performSearch() {
    ...
    await Word.run(async (context) => {
        ...
        let fullSentences: Word.Range[] = [];
        searchItems.items.forEach(range => {
            let sentence = range.getTextRanges(
                [".", "?", "!", ",", ";"]).getFirst();
            sentence.load("text");
            fullSentences.push(sentence);
        });
        await context.sync();

        let $searchResults = $("#search-results").empty();
        fullSentences.forEach(range => {
            range.track();
            let $button = $("<a href='javascript:void(0)'>")
                .text(range.text)
                .click(() =>
                    tryCatch(() => selectAndFormatRange(range)));
            $searchResults.append($button);
        });
    });
}
```

Try it out, midway

If you want to see this midway-completed code, just import the following Snippet ID into Script Lab:
927c81d8e38066cb3461ffea66fe19fb

Note that just as with `object.load(...)` – which is 100% equivalent to `context.load(object, ...)` – the `object.track()` method is an alias for `context.trackedObjects.add(object)`. And, just as with `object.load(...)`, I personally much prefer the `object.track()` form, both because it's shorter, and because IntelliSense guides you into calling this method. When there is no `.track()` method on the object, you'll know that there is no need to ever track it. Admittedly, there is no harm in tracking an object that doesn't need it (the call simply no-op-s), but I still prefer to know whether my objects needs tracking or not, to save myself writing cleanup code.

Speaking of cleanup: This is where you pay the piper for tracking an object. Once you call `object.track()`, the lifetime of the object becomes *your* responsibility. Failure to clean up will result in the host application slowing down over time, proportionally (and perhaps even exponentially) to the number of leaked object. So, while several dozen leaked objects might go reasonably unnoticed – and while certain scenarios, like this one, do require you to reserve the memory for objects that you might eventually use – you should do your best to avoid leaking memory unnecessarily. For example, if the user re-clicks on the "search" button, and you bring down fresh results, there is no way for the user to click on buttons that have now been erased from the DOM. You should use this as an opportunity to clean up, so that you aren't leaking memory unnecessarily (and so that your users don't experience hangs that correlate with using your add-in). Let's do one final edit to the code, this time making sure to store the temporary Range objects into an array, and cleaning up before we fill up the array with new objects:

Being good citizens, and cleaning up

```
let trackedSearchResults: Word.Range[] = [];

async function performSearch() {
    if (trackedSearchResults.length > 0) {
        await Word.run(trackedSearchResults, async (context) => {
            trackedSearchResults.forEach(range => range.untrack());
            trackedSearchResults = [];

            await context.sync();
        })
    }

    let searchTerm = <...>;

    await Word.run(async (context) => {
        let searchItems = context.document.body.search(
            searchTerm, {matchCase: false, matchWholeWord: true});
        searchItems.load("text");
        await context.sync();

        ...
    });
};
```

Note how in the cleanup code, we also use the `Word.run` overload that takes in an object (or in this case, an array of objects). This way, the request for untracking happens on the same context as the one that initiated the tracking. We'll see why it's important in the very next section.

Try it out, the completed code

For a complete and final version of this code, including cleanup, import the

following Snippet ID into Script Lab:
f1f4cd1a4871887c87254f20b86e8340

10.2.2 A common, and infuriatingly silent, mistake: queueing up actions on the wrong request context

For the sake of argument, let's take the code that we had in the previous section, for selecting and formatting the range. But this time, let's *forget* to pass in the `range` object to `Word.run`, and use the normal `Word.run` instead of the overloaded version.

Try it out, by importing the completed snippet from the previous section, and omitting the `range` parameter in the `await Word.run(...)` statement:

What happens when we fail to pass in previously-used object to the 'Word.run' overload?

```
async function selectAndFormatRange(range: Word.Range) {
    // Intentionally (and incorrectly!) forgetting to
    // pass in the `range` object to the `Word.run` overload:
    await Word.run(async (context) => {
        range.font.highlightColor = "yellow";
        range.select();

        await context.sync();
    });
}
```

What happened when the `range` object wasn't passed in?
Answer: *Absolutely nothing*. **No error was thrown, but no action occurred, either**. If you were to step through the code via breakpoints, or add `console.log` statements, you would see that everything *seems* to execute correctly, but the `sync` itself does nothing; it's almost as if it just no-ops. Why?!

For those who read the "*Implementation details*" section, you will remember that the Request Context holds the queue of actions that we want to execute as part of a `context.sync()`. If you browsed through the IntelliSense, you will also have seen that each API proxy object has a `context` property, which points at the request context that it originated from (and which gets passed down from an object to its descendants – e.g., from the root `workbook` object down to a particular `Range` object, in a call to `workbook.getSelectedRange()`). Now, if you combine these two facts, it means that whenever you perform an action on an API object, it gets queued up and recorded *only on its own context*, and flushed only when *its own context is synced*. And so, if you create a new anonymous request context as part of a regular (non-overloaded) `Word.run`, and then use previously-created objects that had been part of an entirely different context, the `sync` does nothing as far as the new actions on the old objects are concerned.

Want an even more infuriating example? Let’s take a look at this code, this time in Excel:

A seemingly-bogus ‘PropertyNotLoaded‘ error

```
let range: Excel.Range;

// Capture the selected range, but don't do
// anything with it yet
await Excel.run(async (context) => {
    range = context.workbook.getSelectedRange();
    range.track();
    await context.sync();
});

// ... some time later (and incorrectly forgetting
//     to use the run overload) ...
await Excel.run(async (context) => {
    range.load("address");
    await context.sync();

    console.log(range.address);
});
```

When you run this code, you will see the following error:

> **PropertyNotLoaded**: The property ‘address’ is not available. Before reading the property’s value, call the load method on the containing object and call `context.sync()` on the associated request context.

Despite knowing better, I’ve still managed to run into this issue occasionally. My mental checklist for debugging these sorts of failures is as follows:

1. “*Am I using the right property name on the right object?*”. To check, I go to the line where I use the property, and see if it’s available in the IntelliSense. It is indeed visible, so clearly a misspelling is not the issue. Move onto the next step.
2. “*Did I break the Promise chain?*”. Nope, all `run` methods `sync` statements are `await`-ed. Move on to the next step.
3. [Scratch head. What *is* the next step?]
4. [Glare silently at the screen, thinking ”*What do you mean that the property isn’t available, can’t you see that I call ‘load‘ and ‘sync‘ above? What else do you want me to do, say ’pretty please’?!*”]
5. […Some time later:] “*Oh yeah, it’s a previously-created object isn’t it?*”. Sure enough, I’m using an object from some past `Excel.run`, inside of a new `Excel.run` and without specifying a property overload. *Whew*.

Fortunately, there is a **very simple best-practice** you can follow, to never end up in this situation again:

BEST PRACTICE: If you are using an existing (previously-created) object inside of a `run` statement, always (ALWAYS!) **be sure to include it as the first parameter in the `.run` overload!** If you’re using *multiple* previously-created objects, that’s OK too, just stick them all into an array and include that array in the overload (the first parameter accepts either a single object or an array).

Think of that first parameter as a way of declaring intentionality, telling the code that “*FYI, I intend to use these other previously-created objects as well, inside this new* `run` *scope*”. Express your intentions, and the runtime will do the right thing if it can, or at least fail with a clear explanation of what went wrong, rather than silently doing nothing!

10.2.3 Resuming with multiple objects

Resuming with multiple objects is just as simple as resuming with one: just pass in the array into the first parameter of the Excel.run (Word.run, etc.) overload! This is what makes it so easy to follow the best-practice from the preceding section, provided you are aware of the issue.

You've actually already seen a subtle use of this in one of the previous code blocks:

A snippet from an example you've already seen before: passing an array of objects into the first parameter of the 'Word.run' overload

```
let trackedSearchResults: Word.Range[] = [];

async function performSearch() {
    ...

    if (trackedSearchResults.length > 0) {
        await Word.run(trackedSearchResults, async (context) => {
            trackedSearchResults.forEach(range => range.untrack());
            trackedSearchResults = [];

            await context.sync();
        })
    }

    ...
}
```

What happens if the objects are from different contexts? You will get an error, as objects from different contexts can't interact with each-other. But again, it's better to have the runtime throw the error at you, rather than having a silent failure. And so again: as a best practice, always pass in *any and all* existing objects that you intend to re-use within the new run scope.

The array-parameter safety net: getting an explicit 'InvalidRequestContext' error, if accidentally combining objects from multiple contexts

```
let range1: Excel.Range;
let range2: Excel.Range;

// Get one Range object
await Excel.run(async (context) => {
    range1 = context.workbook.worksheets
        .getActiveWorksheet().getCell(0, 0).track();
    await context.sync();
});

// ... And, some time later, another:
await Excel.run(async (context) => {
    range2 = context.workbook.worksheets
        .getActiveWorksheet().getCell(5, 5).track();
    await context.sync();
});

// ... Now try to incorrectly resume a context,
//     despite the fact that these are not combinable:
await Excel.run([range1, range2], async (context) => {
    console.log("Will error out before ever getting here");
});
```

For the record, the error reads as follows.

> **InvalidRequestContext**: Cannot use the object across different request contexts.

It's the same exact error as you'd get if you tried mixing contexts in other means (e.g., calling a method that uses an object from a different context). Essentially, the array-accepting `run` overload is just doing extra pre-validation for you, covering something that might or might not have been caught by the runtime otherwise:

The other reason for getting an 'InvalidRequestContext' error: using objects from different contexts in an API call

```
let range: Excel.Range;

// Get one Range object
await Excel.run(async (context) => {
    range = context.workbook.worksheets
        .getActiveWorksheet().getRange("A:A").track();
    await context.sync();
});

// ... And, some time later, call a method that
//     tries to use an object from a different context:
await Excel.run(async (context) => {
    let intersection = context.workbook.worksheets
        .getActiveWorksheet().getRange("3:5")
        .getIntersection(range)
        .load("address");
    await context.sync();

    console.log(intersection.address);
});
```

This also means that if you *do* want to make the above scenario work – using a saved-off object in combination with newly-created objects in a new Excel.run block – you just need to follow the same Best Practice as laid out above, and pass in the existing range object to the second Excel.run invocation. Once you do, the code will run correctly, and output the appropriate intersection address.

10.2.4 Why can't we have a single global request context, and be one happy family?

Glad you asked!

There are two catches to re-using a context, and why we don't just keep one for you for the lifetime of the application.

1. JS runtime memory consumption. So long as there is a reference to a context object (or to one of its children), the memory footprint of the JS code will continue to grow. So for simple "fire and forget" actions, it's better to just create a new context and let it quickly get garbage collected, without having it stick around.
2. Awaiting previous `run`-s. For technical reasons (involving cleanup of tracked objects), a given context can only be used in a single `run` at a given time. This is not necessarily a bad thing (esp. since there can only be so many concurrent requests to the host anyway), but there is still some extra overhead involved in resetting the same context between `run`-s, versus just churning out new contexts and never needing to reset them. So, having a single global context could become a performance bottleneck over time.

This isn't to say you shouldn't use the `run` overloads: they are supremely useful, and are much much easier than trying to manage object cleanup manually (which is why the `Excel.run` concept was created in the first place). Rather, I am merely suggesting that:

- On one hand, you follow the Best Practice, and always use the overload if you're re-using a previously-created object.
- On the other hand, make sure that you are *indeed* using the previously-created object. If you aren't, and it can be a simple fire-and-forget `run` with no external dependencies, you're better off letting the `run` create and destroy its own new request context.

11. Other API Topics

Chapter structure & planned content

Sections outlined here are planned topics, but have not been written (or polished) yet.

- Office.initialize
- Finding the point of failure: Trace Messages
- Recognizing a broken Promise chain. Symptoms: code not executing in order, errors silently swallowed, or InvalidObjectPath error.
- Collection access: `.getItem(key)`, vs. `.getItemOrNullObject(key)`, vs. `.getItemAt(index)` vs. `.items[index]`
- Array properties & complex types (structs / interfaces)

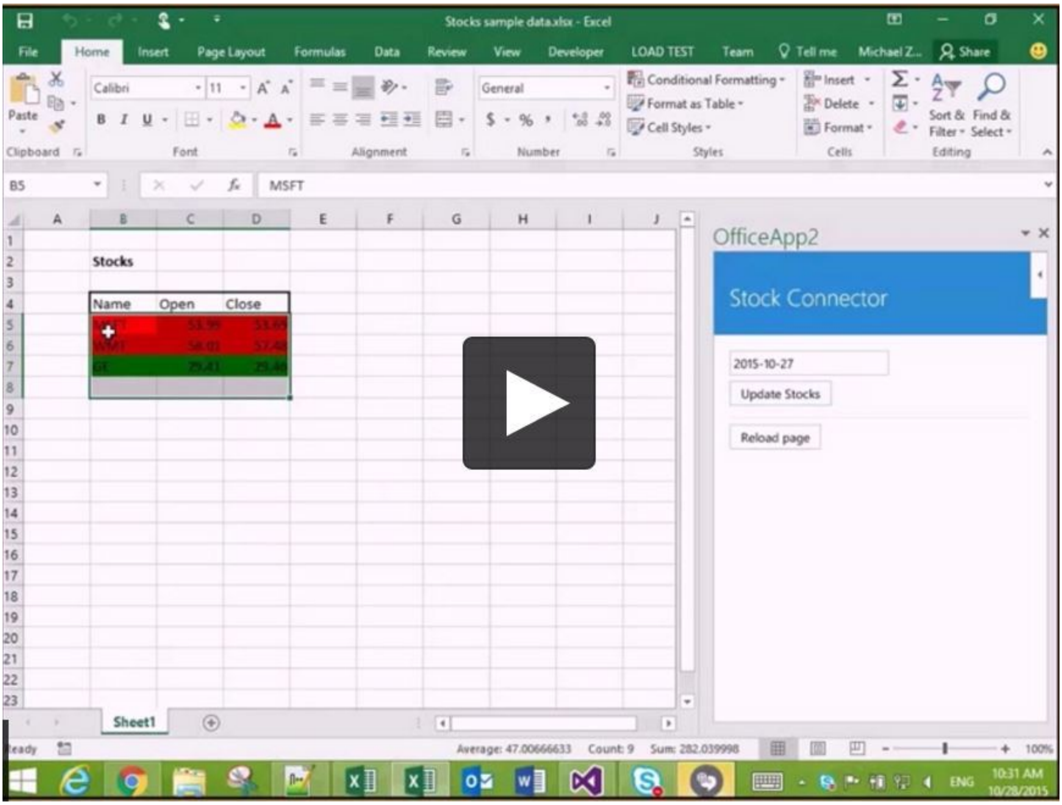
12. The practical aspects of building an Add-in

Chapter structure & planned content

- A video walkthrough of building an Add-in using Visual Studio
- Getting started with building TypeScript-based add-ins
  - Using Visual Studio
  - Using Yeoman generator & Node/NPM
- Debugging: the bare basics
- IntelliSense
- Office versions: Office 2016 vs. Office 365 (MSI vs. Click-to-Run); Deferred vs. Current channels; Insider tracks
- Office.js API versioning
  - The JavaScript files
  - The host capabilities
  - The Beta Endpoint
  - How can you know than an API is “production-ready”?

12.1 Walkthrough: Building an Add-in using Visual Studio

If you’re just getting started and want to use Visual Studio, I highly recommend watching a walkthrough tutorial that I recorded in late 2015: https://channel9.msdn.com/series/officejs/End-to-End-Walkthrough-of-Excel-JavaScript-Add-in-Development.



In the video, I walk through the end-to-end process of building an Office Add-in for Excel: from launching Visual Studio, to writing a bit of JavaScript code that uses the new Excel 2016 APIs, to adding some basic UI tweaks, to talking through the publishing options, debugging, and more.

The video touches on some API topics that are covered in much greater detail in this book – but it also shows the process of creating a project and debugging using Visual Studio, which is crucial for getting started. If you’ve not built an Office Add-in before, I highly recommend the video.

For those looking for written instruction, just on the Visual Studio piece: there is also official documentation for creating a project on https://dev.office.com/docs/add-ins/get-started/create-and-debug-office-add-ins-in-visual-studio.

12.2 Getting started with building TypeScript-based Add-ins

As noted earlier, I firmly believe that TypeScript (as opposed to *plain* JavaScript) offers the premier Add-in coding experience. Depending on how comfortable you are with the emerging web technologies (i.e., Node, NPM, etc), you can either use the Office Yeoman generator to create a typescript-based project, or you can tweak a Visual-Studio-created JS project to convert it to TypeScript.

12.2.1 Using Visual Studio

Currently, the Visual Studio templates for Office Add-ins come only in a JavaScript-based flavor. Fortunately, it does not take much setup to convert the project to TypeScript. To do so, using Visual Studio, create the project *as if it's a regular JavaScript-based Add-ins project*, and then follow the few steps described in this excellent step-by-step blog-post: http://simonjaeger.com/use-typescript-in-a-visual-studio-office-add-in-project/. Once you've done it once, it's you'll see that it only takes a minute or two to do the next time, and is well-worth the trouble.

To get IntelliSense, be sure to add the Office.js d.ts (TypeScript definitions) file, available from https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/office-js/index.d.ts[1]. You can either copy-paste the file manually into your project (and check back periodically to ensure that you have the latest version), or install it via the `@types/office-js` NPM package, which will make the package easier to keep up-to-date.

Note that to use the *async/await* features, you'll need to ensure you have TypeScript 2.1 or higher. For integration with Visual Studio, use Visual Studio 2017 (which comes with TS 2.1+ built-in), or download an extension for Visual Studio 2015 from https://www.typescriptlang.org/index.html#download-links.

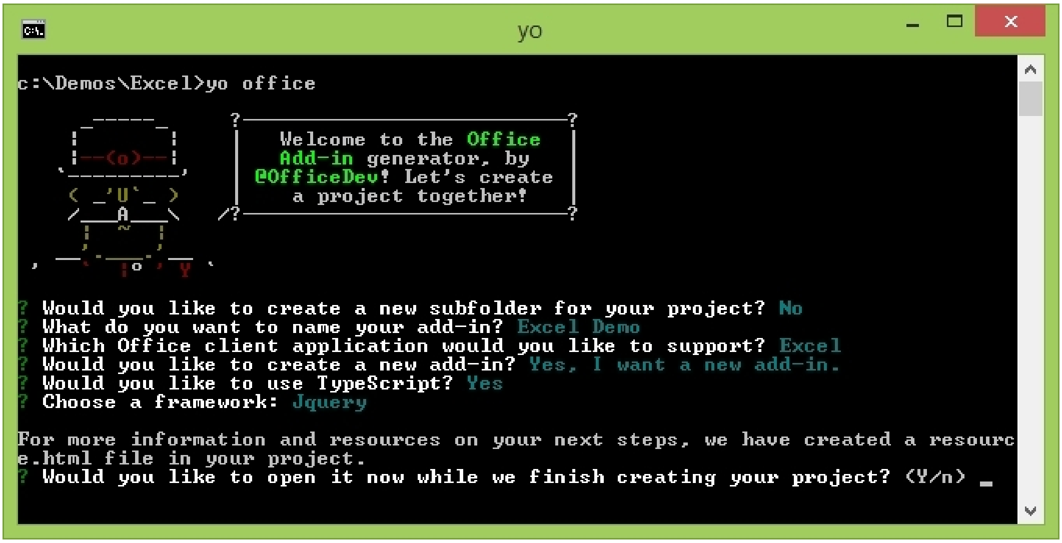
[1] Note that the file (and the rest of the files in the DefinitelyTyped repo) have recently been changed to have the library's main file be called "index.d.ts". This means that some of the older links you might encounter on the web will reference ".../office-js/*office-js.d.ts*", but in reality it should now be ".../office-js/***index.d.ts***"). Likewise, whereas before the path was ".../master/office-js/...", a more recent change added a "types" folder inside of "master". In short, be aware that while the file is certainly on DefinitelyTyped, certain links to it may be broken – so use the latest URL from above.

12.2.2 Using Yeoman generator & Node/NPM

If you are comfortable with Node and NPM (Node Package Manager), you may find it easier to use the Yeoman template-generator for Office instead. Yeoman has been updated in early Feburary 2017 to include TypeScript templates, and to offer a bunch of other goodness (e.g., browser-sync, and the ability to auto-recompile TypeScript sources, etc). Yeoman also offers a way to use Angular 2, instead of the plain html/css/js that comes with the VS template. It requires a tad more setup, esp. the first time around (learning where to trust the SSL certificate and how to side-load the manifest), but the auto-recompilation, browser-sync, and the lightning speed of Visual Studio Code (a web-tailored cross-platform editor, not to be confused with Visual Studio proper) are worth it, if you don't mind going outside the familiar Visual Studio route.

The Add-in documentation team had put together an excellent step-by-step README for how to use the Office Yeoman generator. You can find this documentation here: https://github.com/OfficeDev/generator-office/blob/master/readme.md

Once you've installed the pre-requisites (Node, NPM), and also installed Yeoman and the Office Yeoman generator (`npm install -g yo generator-office`), you can type `yo office`, and Yeoman will guide you through a series of questions:



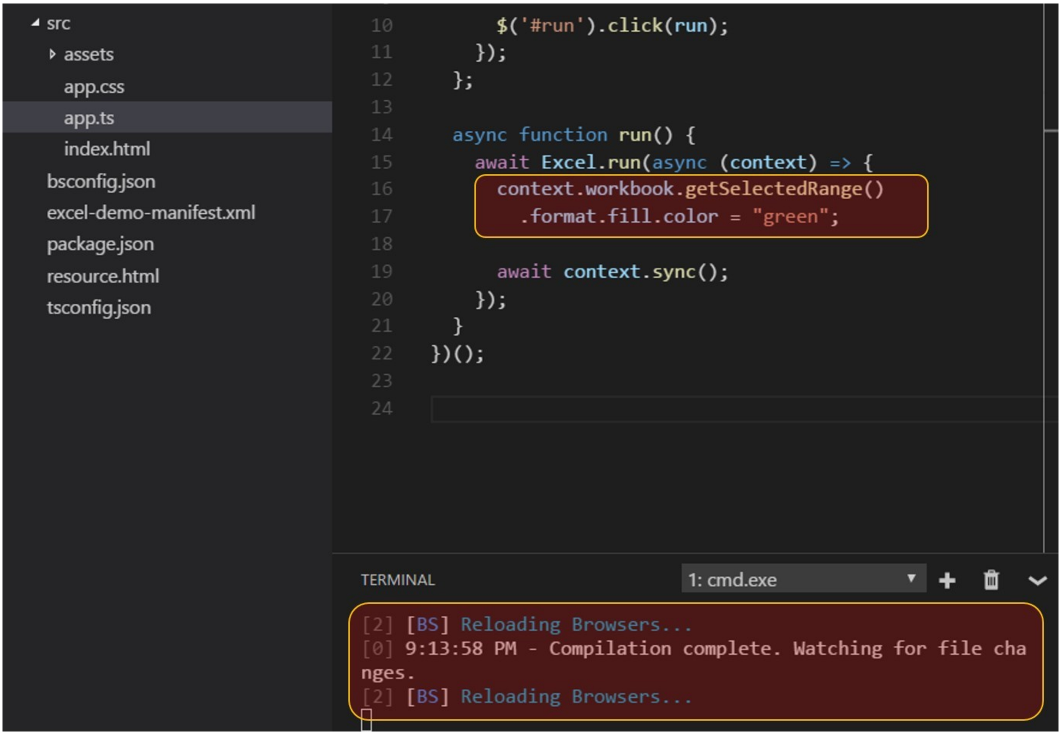
```
c:\Demos\Excel>yo office

     _-----_     ?--------------------------?
    |       |    |   Welcome to the Office  |
    |--(o)--|    |   Add-in generator, by   |
   `---------´   | @OfficeDev! Let's create |
    ( _´U`_ )    |    a project together!   |
    /___A___\   /?--------------------------?
     |  ~  |
   __'.___.'__
 ´   `  |° ´ Y `

? Would you like to create a new subfolder for your project? No
? What do you want to name your add-in? Excel Demo
? Which Office client application would you like to support? Excel
? Would you like to create a new add-in? Yes, I want a new add-in.
? Would you like to use TypeScript? Yes
? Choose a framework: Jquery

For more information and resources on your next steps, we have created a resourc
e.html file in your project.
? Would you like to open it now while we finish creating your project? (Y/n) _
```

When it's done, you will have a project set up, complete with all the goodness of the NPM world and a couple of config files that wire the magic together. Run `npm start`, and the web portion of the Add-in will launch. Once you've done this, you need to do just a couple things:

1. Trust the SSL certificate. See https://github.com/OfficeDev/generator-office/blob/master/src/docs/ssl.md
2. Side-load the Add-in (via the Add-in manifest) into the Office application. The manifest will be created alongside all the rest of your project's files, in a file called `<your-project-name>-manifest.xml`
   Sideloading is very easy for Office Online, and somewhat more cumbersome on the Desktop, but just follow the instructions or step-by-step video here: https://aka.ms/sideload-addins

When you're ready to write some code, open the folder in your favorite editor. **Visual Studio Code** is an absolutely excellent lightweight editor, which I have been using as a companion (and often, my go-to tool) for web things. You can even open a terminal straight within VS Code (use **ctrl** + ‘ [backtick])!

The really cool thing about using the Yeoman template and the browser-sync functionality is that as soon as you make a change to the code and save, your code gets automatically re-compiled and reloaded!



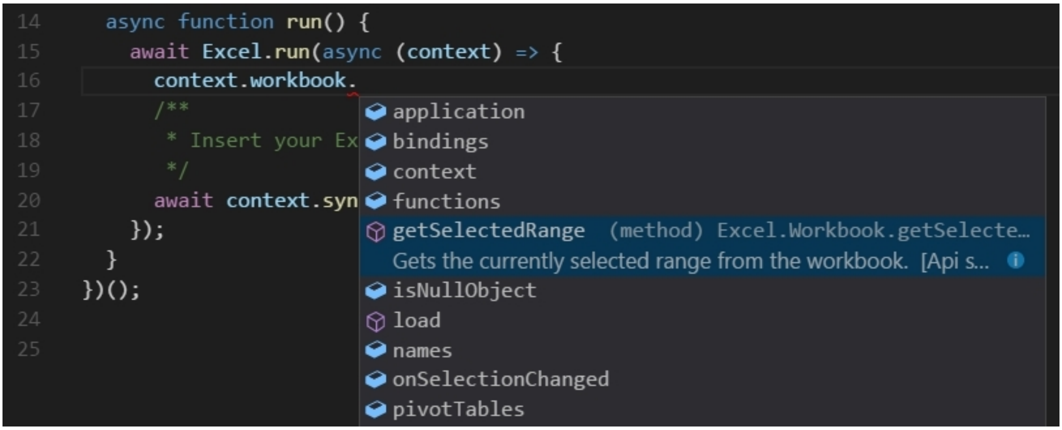
```
   $('#run').click(run);
  });
 };

 async function run() {
  await Excel.run(async (context) => {
   context.workbook.getSelectedRange()
    .format.fill.color = "green";

   await context.sync();
  });
 }
})();
```

```
[2] [BS] Reloading Browsers...
[0] 9:13:58 PM - Compilation complete. Watching for file cha
nges.
[2] [BS] Reloading Browsers...
```

Note the auto-re-compilation of the files. The Add-in, too, will automatically re-load.

By the way, if you do not see IntelliSense when you type in something like "`Excel.`" or "`Word.`" or when you try to modify something inside of the `Excel.run` or `Word.run` block, please add a reference to the Office.js d.ts (TypeScript definitions) file. To add it, simply run

```
npm install @types/office-js
```



```
  async function run() {
    await Excel.run(async (context) => {
      context.workbook.
      /**
       * Insert your Ex
       */
      await context.syn
    });
  }
})();
```

application
bindings
context
functions
getSelectedRange (method) Excel.Workbook.getSelecte...
Gets the currently selected range from the workbook. [Api s...
isNullObject
load
names
onSelectionChanged
pivotTables

IntelliSense is back!

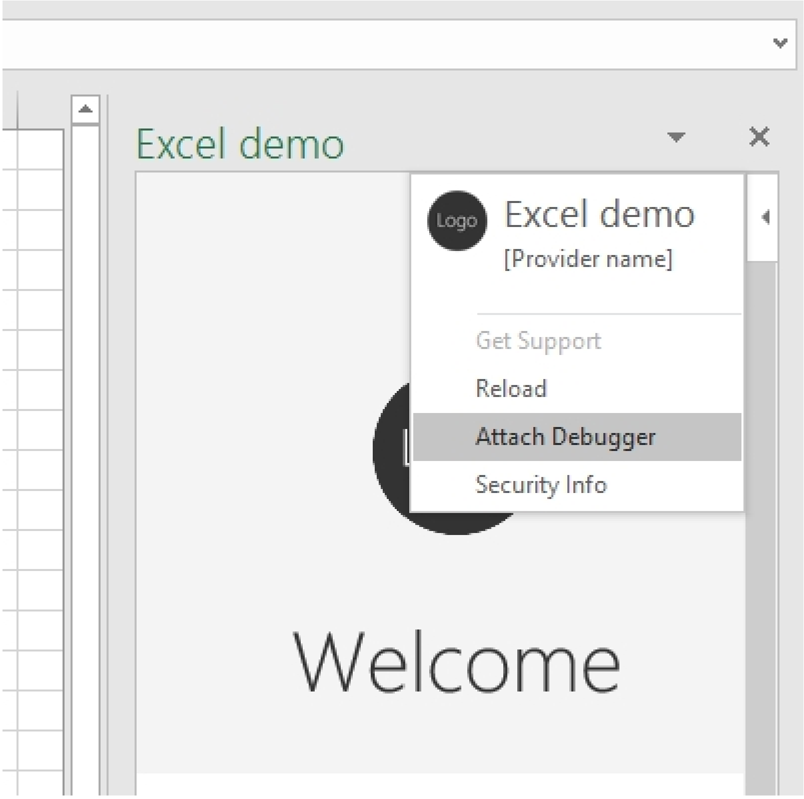
12.3 Debugging: the bare basics

On Windows, if you are using the Visual Studio templates for Office Add-ins, the debugger will get automatically attached when you press Run (F5). It can't get simpler than that…

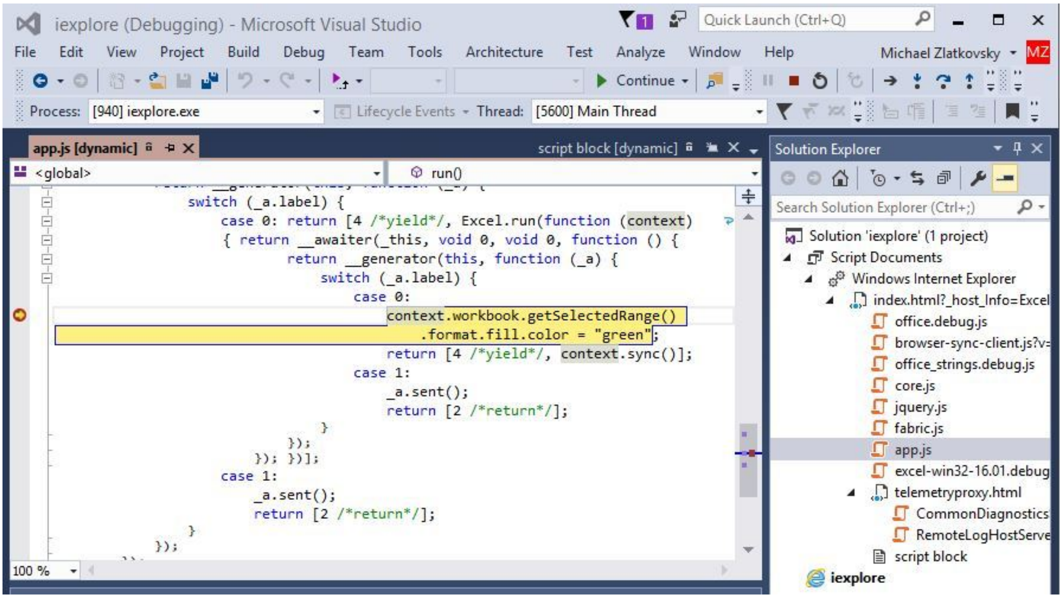
If you are *not* using Visual Studio (for example, you're **wanting to debug a Script Lab snippet** or are using **a template created by the Yeoman-generator** and edited inside of VS Code), it's no problem, either – but your best bet is *still* to download and install Visual Studio, and use it for the debugger component even if you are happy with your editor setup[2]. To attach the VS debugger to a running Add-in, you can either do an "Attach to Process" from within VS (a cumbersome process involving first closing down all Internet Explorer instances, and then attaching the debugger to the remaining `iexplorer.exe` processes) – OR you can use a very cool recent feature[3] that makes the process much simpler. Simply click on the *Personality* menu at the top right of the Add-in, and select "Attach Debugger", as follows:

[2] I have heard that some folks have had success with Windows 10's standalone F12 tool. So, if you are opposed to having Visual Studio on your computer (and don't have it free courtesy of an MSDN subscription, BizSpark, etc.), you probably do have this option – but it won't be integrated with the "Attach Debugger" menu, and I'm not 100% sure if it's quite as powerful as its VS counterpart.

[3] How recent is recent? You will need an Office 365 install of Office for Window, build number `77xx.xxxx` or later. See https://dev.office.com/blogs/attach-debugger-from-the-task-pane for more info.



A little while later, Visual Studio will launch, with your web assets in the Solution Explorer. Simply find the script file you’re interested in, and add a breakpoint at the desired spot.

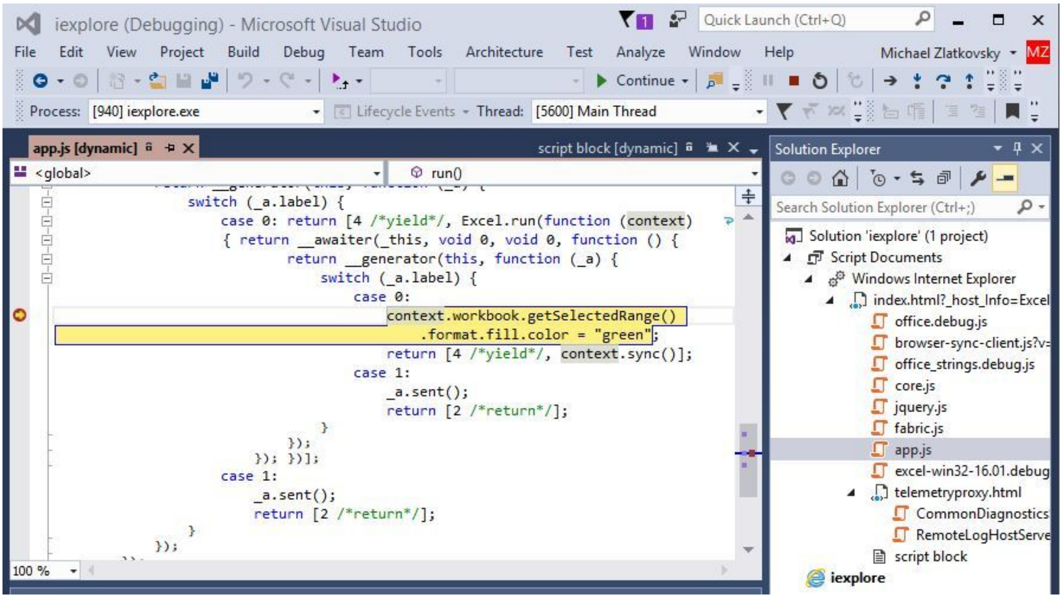

iexplore (Debugging) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help
Michael Zlatkovsky
MZ
Continue
Process: [940] iexplore.exe
Lifecycle Events
Thread: [5600] Main Thread
app.js [dynamic]
script block [dynamic]
<global>
run()
100 %
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'iexplore' (1 project)
Script Documents
Windows Internet Explorer
index.html?_host_Info=Excel
office.debug.js
browser-sync-client.js?v=
office_strings.debug.js
core.js
jquery.js
fabric.js
app.js
excel-win32-16.01.debug
telemetryproxy.html
CommonDiagnostics
RemoteLogHostServe
script block
iexplore

```
    switch (_a.label) {
        case 0: return [4 /*yield*/, Excel.run(function (context)
        { return __awaiter(_this, void 0, void 0, function () {
                return __generator(this, function (_a) {
                    switch (_a.label) {
                        case 0:
                            context.workbook.getSelectedRange()
                                .format.fill.color = "green";
                            return [4 /*yield*/, context.sync()];
                        case 1:
                            _a.sent();
                            return [2 /*return*/];
                    }
                });
            }); })];
        case 1:
            _a.sent();
            return [2 /*return*/];
    }
});
```

An active breakpoint. By the way, note how the TypeScript 'async/await' gets compiled down to pretty strange-looking code, to create the illusion of synchronous commands!

DOM Explorer

Regardless of whether you attach the debugger through Visual Studio's "Run" or through the Add-in, there is another very useful tool to use while the Add-in is under the debugger. Tucked away like a hidden gem, the DOM (Document Object Model) Explorer allows you to view and modify the HTML content of your Add-in, *live*. Personally, I find this tool to be indispensable when debugging an Add-in layout issue. To launch it, simply choose the "Quick Launch" textbox at the top-right of VS, enter in "DOM", and select the "Show All" match.



Process: [940] iexplore.exe

Thread: [5600] Main Thread

```
    switch (_a.label) {
        case 0: return [4 /*yield*/, Excel.run(function (context)
        { return __awaiter(_this, void 0, void 0, function () {
                return __generator(this, function (_a) {
                    switch (_a.label) {
                        case 0:
                            context.workbook.getSelectedRange()
                                .format.fill.color = "green";
                            return [4 /*yield*/, context.sync()];
                        case 1:
                            _a.sent();
                            return [2 /*return*/];
                    }
                });
            }); })];
        case 1:
            _a.sent();
            return [2 /*return*/];
    }
});
```

Debugging on other platforms

To debug in Office Online, just use the browser's F12 tools directly. I personally am partial to the Chrome Dev Tools, but the F12 tools of Internet Explorer, Edge, and Firefox will all offer similar functionality (breakpoints, variable watches, a console, and the ability to manipulate the HTML/CSS in real-time).

To debug on the Mac or on an iPad, see https://dev.office.com/docs/add-ins/testing/debug-office-add-ins-on-ipad-and-mac

12.4 IntelliSense

To program without autocompletion & parameter hints might be possible in some langauges & frameworks… but Office.js is absolutely *not* one of them. The APIs are varied and deeply hierarchical, and often contain small subtleties that are explained in the IntelliSense description or parameter hints, but that would be difficult to guess. In my opinion, IntelliSense is an absolute must for having a successful and enjoyable Office.js experience.

Fortunately, making IntelliSense work is pretty easy.

- For JavaScript: it should "just work" inside of a Visual-Studio-created Office Add-in project. But in case it doesn't (and especially on Visual Studio 2017), see the following article.
- For TypeScript:
  - If you are using the latest public APIs, just install the `@types/office-js` NPM package, or get the file off of DefinitelyTyped.
  - If you are using the BETA channel, grab the `dist/office.d.ts` file off of the `beta` branch. https://github.com/OfficeDev/office-js/blob/beta/dist/office.d.ts

… And of course, you can always try out the APIs – IntelliSense and all – in Script Lab. See "*Script Lab: an indispensable tool*" from earlier in the chapter.

12.5 Office versions: Office 2016 vs. Office 365 (MSI vs. Click-to-Run); Deferred vs. Current channels; Insider tracks

To develop and distribute Add-ins that use the new Office 2016 API model, you need either Office 2016 or Office 365 (the subscription-based superset that includes all 2016 features). This seems reasonably straightforward, but the devil is in the details.

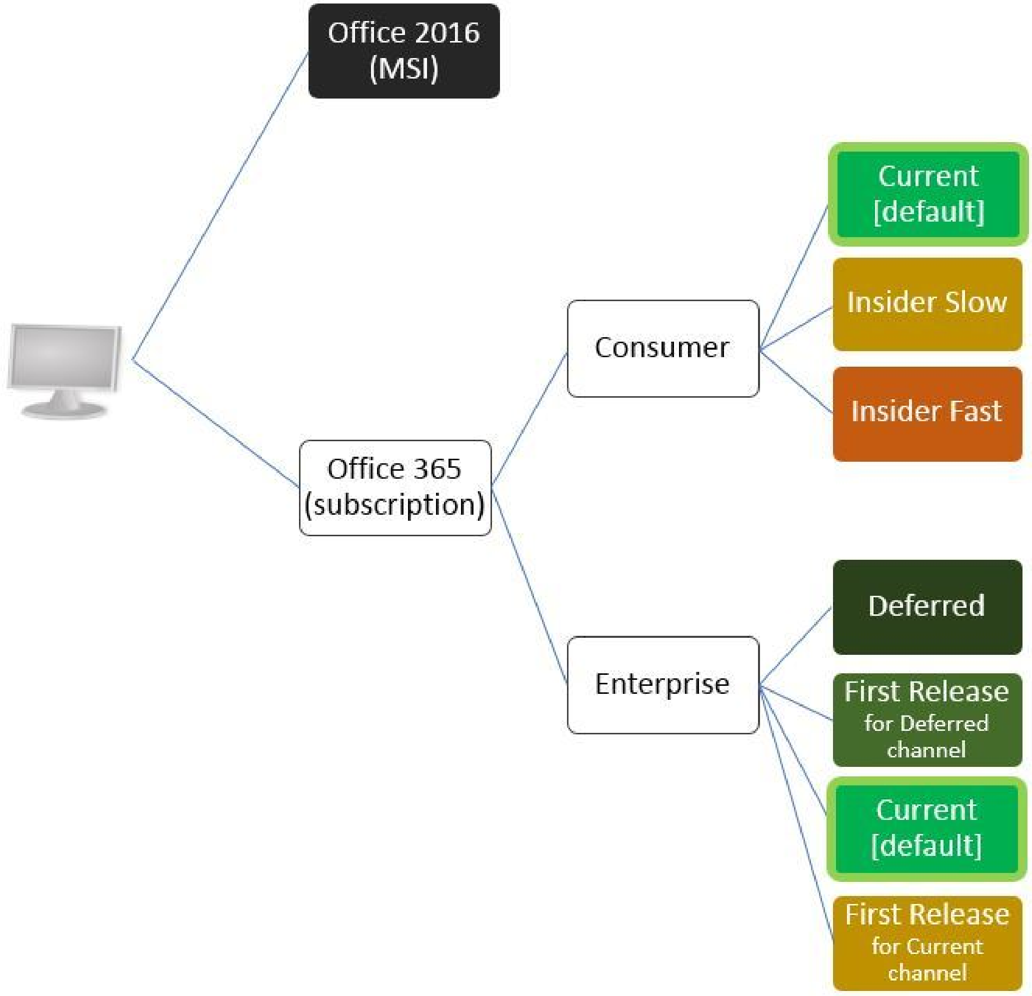
The golden path

The simplest possible case is when both you (the developer) and your end-users have the latest-and-greatest versions of Office 365. Certainly, for development / prototyping needs, having Office 365 with the latest updates will be simplest. But if you are an ISV (Independent Software Vendor) and hence have no control over what version your customers are running; or if you work inside of an enterprise that might not be on the bleeding edge, that's where understanding Office versions becomes important.

Why you should care

Different versions (and categories of versions) offer different API surface areas. For example, Office 2016 RTM only offered the first batch of the new 2016+ wave of Excel and Word APIs; those APIs have been greatly expanded since then (see the discussion on API versioning in the next section) . Likewise, other functionality – most notably, Add-in Commands (ribbon extensibility) and the ability to launch dialogs – were not present in the original RTM version.

In the next few pages, I will describe the different installation possibilities. It may help to bear the following image in mind:



Office 2016 vs. Office 365

The first place where the API surface-area forks off is at the split between the MSI-based installation of Office 2016, and the subscription-based (sometimes called “click-to-run”) installation of Office 365.

Let’s pause for a second to talk about Office 365, as I’ve seen some confusion about the term. Wikipedia explains it nicely:

> Office 365 is the brand name Microsoft uses for a group of software and services subscriptions, which together provide productivity software and related services to subscribers. For consumers, the service allows the use of Microsoft Office apps on Windows and macOS, provides storage space on Microsoft's cloud storage service OneDrive, and grants 60 Skype minutes per month. For business users, Office 365 offers service plans providing e-mail and social networking services through hosted versions of Exchange Server, Skype for Business Server, SharePoint and Office Online, integration with Yammer, as well as access to the Microsoft Office software.

So, for those coming from the SharePoint world: yes, SharePoint Online is part of an Office 365 subscription, as are the Office Online in-browser editors that come with it. But, it is not the *only* part of the subscription. Getting access to the *same desktop/mac Office programs that you know and love* is also part of that same subscription (as is getting iOS and Android versions of those Word, Excel, PPT, etc. programs).

Now, back to the APIs: if you have Office 2016 (non-subscription), you will *only* have the initial set of the new wave of Excel and Word APIs (`ExcelApi 1.1` and `WordApi 1.1`). Or to put it even clearer: you will only have the *initial set of extensibility functionality* – period. So in addition to missing improvements to the Excel and Word APIs, you will also lack other add-in functionality like the ability to customize the Ribbon or launch dialogs.

It's also worth noting that the original RTM offering of the APIs did have some bugs. Some were more innocent than others[4], but – in my personal opinion – I would treat RTM as more of *the start of a journey* into rich host-specific APIs, rather than a destination of its own.

[4] There are a number of bugs that come to mind. On the Excel side, for example, reading back values (even without manipulating the document) would blow the undo stack – not a catastrophic issue, but irksome all the same. On the Word side, there were some issues with document-identity when accessing items in a collection (e.g., paragraphs), whereby the proxy object remembered *its index in the collection*, but not the actual document entity that it belonged to; and so, if the document was manipulated, the proxy object (paragraph) would now point at the wrong item.

So again: Office 2016, from an API / extensibility standpoint, is frozen in time… frozen to the functionality that was there when it shipped in September 2015. (For those unclear on how a "2016"-branded product could have shipped in 2015, I guess it's a bit like buying next year's car models. Or, perhaps, `2015 + 3/4-of-a-year` rounds to `2016`…)

Meanwhile, Office 365 means "subscription". This translates to being on the latest-and-greatest stable build (where "stable" for enterprise might be a build that's a few months old; more on that below).

If you want access to the latest-and-greatest API functionality – which as a developer, you absolutely do – you *must* be on a subscription-based installation of Office, rather than the frozen-in-time Office 2016 MSI installation. Moreover, for most of the new functionality, you probably want your customers to be on a subscription-based installation, as well.

Office 365 flavors for the Consumer

The Consumer (non-business) versions of Office 365 include **Office 365 Personal** and **Office 365 Home** (with the only difference between the two being the number of active devices on the subscription – 1 PC or Mac, 1 Tablet, and 1 Phone, as opposed to 5 of each). There is also **Office 365 University**, which is the same as Personal, but offers activating on *two* devices rather than one. For all three, the difference is merely the cost of the plan and the number of devices supported; they are all 100% equivalent from an API & functionality standpoint.

The Consumer versions of Office 356 are updated each month, with the updates installed silently and automatically. Thus, consumer versions of Office 365, provided the computer is connected to the internet, will always have access to the latest-and-greatest functionality. The default is the "Current" channel (i.e., what is publicly available worldwide), but the adventurous user (developer) can also opt in to be on one of the *Insider* tracks. The Insider tracks come in two flavors, Insider Fast and Insider Slow, with the *Fast* being really-bleeding-edge, and *Slow* being a few weeks behind, anchored around more stable builds. In both cases, they let you preview the forthcoming functionality a month or two ahead of the general public. For developers, this can be particularly useful for trying out the latest APIs ahead of your customers, allowing you to deliver new functionality as soon as it's publicly-available on your customers' machines. Combined with using the Beta CDN for Office.js, it can also let you provide real-time API feedback back to the team, before the APIs get cemented and go live! To become an Insider, see https://products.office.com/en-us/office-insider.

Office 365 flavors for Enterprise

For users of the enterprise / business flavor of Office 365, there are also a number of options (typically handled by the IT administrator). Like with the Consumer versions, there is a “Current” channel (latest-and-greatest stable build, updated monthly) – and similarly, there is a “First Release for Current channel”, which essentially is the same as the “Insider” builds in the consumer version.

However, risk-averse enterprises may also choose to be on a Deferred channel, which updates once every four months instead of once every month. Moreover, these enterprises can also stay on the Deferred channel for four or even eight months, before jumping ahead to a newer build. Thus, a business on the Deferred channel may still be a fair bit behind the developer in terms of what API functionality is available (though less behind than someone on the RTM build of Office 2016).

Office on other platforms (Mac, iOS, Online)

For non-PC platforms, there is also a span of time before different functionality lights up. This is sometimes dependent not just on the *delay* between something being code-complete and getting in front of customers’ hands (i.e., the difference between Insider and Current and Deferred), but also on the order in which functionality gets implemented on these platforms. For the Excel APIs to date, I have seen them light up on most platforms at roughly the same time; for Word, the Desktop has generally been ahead of Online. For the non-API functionality (i.e., dialogs, ribbon extensibility), these have also generally come to the Desktop first, followed by Online and Mac. The different speeds of implementation is why it’s important to keep in mind not just Office host versions, but also API versions and Requirement Sets – the subject of the very next section.

12.6 Office.js API versioning

There are several aspects to the versioning of the Office.js library.

12.6.1 The JavaScript files

First, there is versioning of the actual **JavaScript source files**. Fortunately, this part is pretty simple: you **always want the latest and greatest of the production Office.js**, which is conveniently obtainable through the CDN: https://appsforoffice.microsoft.com/lib/1/hosted/Office.js[5].

In addition to the CDN, the production Office.js files are also shipped as a NuGet package to allow corporate firewalled development or offline development. That said, the NuGet may lag several weeks behind the CDN, and any Store-bound add-ins are *required* to reference the production CDN location anyway. So, in terms of Office.js versions, there isn't really much to the versioning of the actual JS files: by far and away the best choice is to reference the evergreen, frequently-updated, always-backwards-compatible, Store-required CDN version.

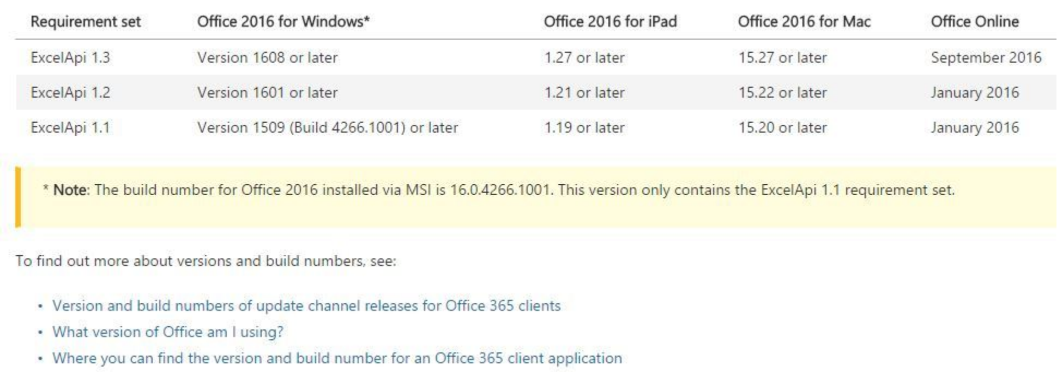
12.6.2 The host capabilities

The more interesting bit for versioning are the **actual API capabilities that are offered by each host**. Just because you have the latest and greatest JavaScript does not mean that older clients will be able to make use of all of it. While some of your customers might be on the latest-and-greatest versions of Office, others won't be! In particular, Office 2016 RTM will always only have access to the original set of Office 2016 APIs – period. And even customers who own subscription versions of office (Office 365 / click-to-run) might still be on older versions of the supposedly-evergreen version, depending on whether

[5] Note that even though your HTML page will reference `Office.js`, in practice that file is only a loader – it will then go and load the *actual* host-specific files, such as https://appsforoffice.microsoft.com/lib/1/hosted/excel-web-16.00.js. So if, for whatever reason, you are looking at the text headers of the JS files, note that the interesting version number is that of the host-specific file, *not* of Office.js itself.

they are on the *Current Channel* or the *Deferred Channel* of Office 365[6]. This intersection of native-application-versioning to javascript-versioning is where things get trickly.

The solution to this complexity (albeit complex in its own right) is *Requirement Sets*. For example, if you look at the Excel API sets documentation[7], you will see that the 2016 wave of Excel APIs has had three versions as of December 2016: 1.1, 1.2, and 1.3. ExcelApi 1.1 was what shipped with Office 2016 RTM in September 2015; 1.2 shipped in early March 2016; and 1.3 shipped in October of 2016. Each API set version has a corresponding Office host version that supports this API set. The version numbers are listed in the table, and there are links below the table to find a mapping from build numbers to dates.



Requirement set	Office 2016 for Windows*	Office 2016 for iPad	Office 2016 for Mac	Office Online
ExcelApi 1.3	Version 1608 or later	1.27 or later	15.27 or later	September 2016
ExcelApi 1.2	Version 1601 or later	1.21 or later	15.22 or later	January 2016
ExcelApi 1.1	Version 1509 (Build 4266.1001) or later	1.19 or later	15.20 or later	January 2016

* **Note:** The build number for Office 2016 installed via MSI is 16.0.4266.1001. This version only contains the ExcelApi 1.1 requirement set.

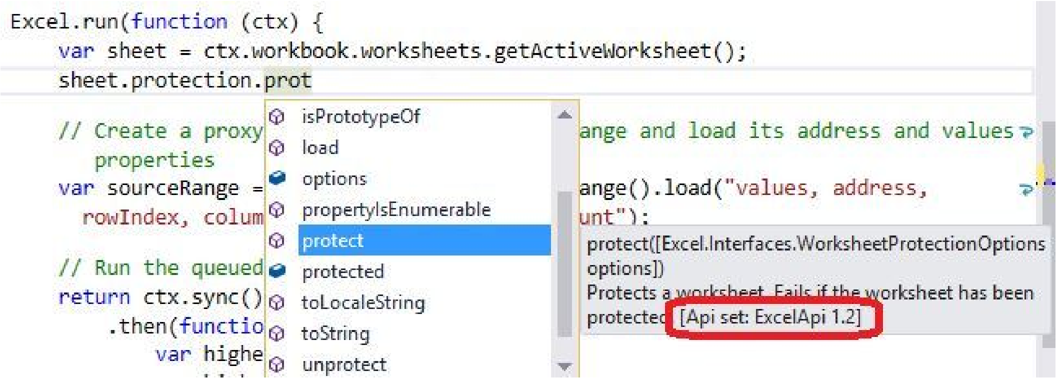
To find out more about versions and build numbers, see:

- Version and build numbers of update channel releases for Office 365 clients
- What version of Office am I using?
- Where you can find the version and build number for an Office 365 client application

Each of the API set versions contain a number of fairly large features, as well as incremental improvements to existing features. The topic for each requirement set, such as the link above, will provide a detailed listing of each of those features. And as you're programming, if you are using the JavaScript or TypeScript IntelliSense, you should be able to see the API versions for each of your APIs displayed as part of the IntelliSense:

[6] Being on a *Deferred Channel* is an option offered to business customers, allowing them to lock in to older versions and defer the monthly updates until they've had sufficient time to test out the new features. See https://technet.microsoft.com/en-us/library/mt455210.aspx.

[7] https://dev.office.com/reference/add-ins/requirement-sets/excel-api-requirement-sets



```
Excel.run(function (ctx) {
    var sheet = ctx.workbook.worksheets.getActiveWorksheet();
    sheet.protection.prot

    // Create a proxy                      ange and load its address and values
       properties
    var sourceRange =                      ange().load("values, address,
      rowIndex, colum                      unt");

    // Run the queued
    return ctx.sync()
        .then(functio
            var highe
```

protect([Excel.Interfaces.WorksheetProtectionOptions options])
Protects a worksheet. Fails if the worksheet has been protected [Api set: ExcelApi 1.2]

You can use the requirement set in one of two ways. You can declare in the manifest that "I need API set ExcelApi 1.2, or else my add-in doesn't work at all" – and that's fine, but then of course you aren't able to service older hosts, and so your add-in won't even show up there. Alternatively, if you add-in could *sorta* work in a 1.1 environment, but you want to *light-up* additional functionality on newer hosts that support it, you can use the manifest to declare only your minimal API sets that you need (e.g., ExcelApi 1.1), and then do runtime checks for higher version numbers via the `isSetSupported` API.

For example, suppose that you are exporting some data to a new sheet, and you'd ideally like to autofit the column width – but this is only available in ExcelApi 1.2. Rather than block the add-in outright from running on an older host, you can do a light-up scenario on newer hosts that support the API, and skip over the functionality otherwise:

Light-up functionality for a newer API set

```
Excel.run(context => {
    let data = [
        ["Name", "Phone number"]
        ["Joe Doe", "425-123-4567"],
        ...
    ]
    let newSheet = workbook.worksheets.add();
    let dataRange = newSheet.getCell(0, 0)
        .getResizedRange(values.length - 1, values[0].length - 1);
    dataRange.values = data;

    if (Office.context.requirements.isSetSupported("ExcelApi", 1.2)) {
        dataRange.format.autofitColumns();
    }

    await context.sync();
})
```

Important: Version numbers are *per API SET*

To be very clear: Version numbers such as `ExcelApi 1.3` are relative *to the API set iself*, but has nothing to do with the version numbers of Office.js or of other API sets. The different requirement sets move at completely different speeds (which makes sense – you wouldn't want Excel APIs to be delayed by a few months, simply so that the Excel's 1.3 set can ship at the same time as Word's 1.3 set).

Because of this, Office.js is an amalgam of different API Set version numbers at any given time. So, while there is definitely such a thing as `ExcelApi 1.3`, ***there is no Office.js 1.3***. It's always the single evergreen Office endpoint, and a variety of versioned API Sets that comprise it.

12.6.3 The Beta Endpoint

In addition to the production CDN, there is also a **beta** endpoint; with the same URL but with "beta" in place of "1". Thus, the full Beta URL is https://appsforoffice.microsoft.com/lib/**beta**/hosted/Office.js

Before describing this beta endpoint, it's worth explaining the general lifecycle of an API. It goes something like this:

1. Before any implementation starts, requirements & feature requests are gathered via a variety of channels (UserVoice, StackOverflow, conversations with partners, etc).
2. From those requirements, a number of feature areas (groups of related API functionality) are chosen to be the team's focus for the next release

(where "release" is an increment to an API set, generally spaced out every 3-4 months). The APIs are then designed and iterated on.
3. When the design is ready, and roughly at the same time as implementation begins, the API specs are posted as part of the "Open Spec" process (https://dev.office.com/reference/add-ins/openspec), and feedback is gathered from members of the community (and incorporated into the design – the sooner we get the feedback, the better!). The APIs might also be expanded to include related functionality, or tweaked based on implementation constraints.
4. The implementation is complete; but the actual release of the full API set might be another couple months away. This is where the **beta** process comes in: for folks who are on the Office Insider Fast builds, they may be able to get the APIs and experiment with them a couple months ahead of their production rollout. This is useful to developers so that they can plan ahead and be ready with new features once the API makes it out publicly; and it can also act as a catalyst for providing feedback about the API from real-life usage, rather than just imagining it in spec form.
5. Some time later, the API becomes part of the next wave of API sets, and is shipped to the world.

To use the beta endpoint, simply change out the script reference, and try out some APIs that you see in the open spec. Some might be ready; others might not. If you encounter a JavaScript runtime error to the tune of a nonexistant object or function, it's not part of the Beta endpoint yet; if the JavaScript is OK but you get an "`ApiNotFound`" error, you do not have a sufficently recent build (or perhaps no such build exists, yet). Desktop, and sometimes Office Online, are the first two channels where the APIs are made available during the beta stage; the other platforms migth only receive the APIs once they roll into production.

There are a few caveats to the beta endpoint:

- It is inherently more experiemntal in nature, and should *not* be used for production.
- Any new APIs therein are effectively in preview mode, and might change at any time. Possible changes include renaming, postponing (moving to a later API set), or having the API be removed or reworked altogether.

- While an API is part of an actively-worked-on (not-yet-shipped) API set version, the `isSetSupported` for said version will return `false` (regardless of production or beta JavaScript), so you won't be able to perform this check for the new APIs, until they make it into production.

All this being said: As you become more familiar with the platform, and especially as you see features that you care about become open-spec-ed, I strongly encourage you to participate in the feedback process, and to use the beta endpoint to validate that the design meets your needs. And as always, questions or feedback on StackOverflow about the APIs – whether production or beta – are always welcome.

Tip: writing `isSetSupported` for the Beta endpoint

As noted earlier, APIs in beta will not show `isSetSupported` as `true` until the functionality is already production-ready (see next section). But on the other hand, what if you want to write the code just as you would for production, without changing a line (i.e., without needing to go back and uncomment the previously-commented-out `if` statements?)

My personal trick for this: go ahead and write your `isSetSupported` checks and `if` statements just as you would for a regular production API. Go to your main HTML page (which ideally you have two copies of – one for production, and one for development, where you can reference the Beta CDN) and write in the following script tag right beneath your beta CDN reference (let's assume that the API set you care about is `ExcelApi`; and you effectively want *any* new APIs to show up as if they are fully supported, for testing purposes):

```
<script>
  (function() {
    var originalIsSetSupported = Office.requirements.isSetSupported;
    Office.requirements.isSetSupported = function (apiSet, version) {
      if (apiSet === 'ExcelApi') { return true; }
      return originalIsSetSupported(apiSet, version);
    };
  })();
</script>
```

Essentially, the script above will return `true` for any API at all that is part of the API set that you want the latest-and-greatest beta functinality of (in this case, `ExcelApi`), and otherwise will redirect to the original `isSetSupported` function.

12.6.4 How can you know than an API is "production-ready"?

An API is production-ready when you look at the IntelliSense, see what API set version the API is supposed to be a part of, do `isSetSupported` on that version number, and see it return `true`.

This moment should roughly correspond with:

- Seeing the API listed in the documentation, no longer under an Open Spec.
- Seeing its IntelliSense listed in a public place like DefinitelyTyped (https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/office-js/index.d.ts)

Note that some platforms might be ahead of others, with Desktop generally leading. So just because `isSetSupported` returns `true` on Desktop, doesn't mean that it will necessarily return `true` on the Mac at the very same moment (though it generally will, within a month or two's time). But the different cadence of implementation is precisely why `isSetSupported` (and/or manifest-based requirements specification) is needed in the first place. As long as you surround any light-up functionality with `isSetSupported`, or add the manifest requirement, you shouldn't need to worry about what APIs are made available when; when the functionality becomes available on the given platform, your add-in will "just work" there.

13. Appendix A: Using plain ES5 JavaScript (no `async/await`)

Chapter structure

13.1 Passing in functions to Promise `.then`-functions

JavaScript-only

This section is thematically related to the “***Promises primer***“ section, coverting a technique that is very useful to **for folks writing in pure JavaScript**, but unnecessary for folks who are coding in TypeScript and with `async/await`. TypeScript and its async/await syntax don’t need to use a `.then`, so the issue goes away on its own.

As seen in section “***Promises, try/catch, and async/await***“, a typical *JavaScript* use of Promises might look like this:

A typical *JavaScript* Promise chain, with in-lined anonymous functions

```
performAsyncOperation()
    .then(function() {
        // ...
        // some synchronous operations
        // ...

        // return eiher a Promise or a regular value
        return xyz;
    })
    .then(...)
    // ... {rinse and repeat}
    .catch(...)
```

In the above, the function that’s passed in to the first `.then` is known as an *anonymous* function – that is, a function whose body has been *inlined* into the code, as opposed to being isolated out into its own standalone function. Depending on the size of the anonymous function, and especially if that function continues to evolve, you might find it more convenient to isolate out the logic into its own separate *named* function, as follows:

The isolated subroutine

```
function myFunction() {
    // ...
    // some other operations
    // ...
    // return xyz;
}
```

There are two ways to make use of this function, now that it's been isolated out. One way is to still keep the anonymous function, but do a one-liner invocation of the named function from within the anonymous one (and not forget the `return` statement!):

A JavaScript Promise chain, doing a one-liner invocation of ‘myFunction‘

```
performAsyncOperation()
    .then(function() {
        return myFunction();
    })
    .then(...)
    .catch(...)
```

There is nothing wrong with doing the above – assuming you remember the `return` statement, that is. But if you pause to analyze the code, you'll notice that all you're doing is creating an *anonymous* function which takes in no arguments, and then invoking a *named* function that *also* takes in no arguments[^arguments]. Functions are first-class citizens in JavaScript, so rather than creating a new anonymous function that all it does is return the value of a function call to another function, you can use the function directly in the `.then` statement:

Invoking a named function from within the ‘.then‘

```
performAsyncOperation()
    .then(myFunction)
    .then(...)
    .catch(...)
```

Note the lack of the typical function-invocation parentheses following `myFunction`. You are *passing the function in* as an argument to the `.then`, you are ***not*** executing the function at this point!

Personally, I find the above to be simpler to read, quicker to type, and less error-prone when dealing with asynchronous functions (where forgetting a “return” statement would lead to a broken promise chain, as described in an earlier section: “*Chaining Promises, the right way*”).

13.2 JavaScript-only version of the canonical code sample

JavaScript-only

This section reflects a *JavaScript* (no-TypeScript) version of the “***Canonical code sample***“ (see the original section for information on the scenario that this code addresses)

The major difference to note is that instead of using `async/await`, the code uses a series of `.then`-s and a final `.catch`. This is certainly possible, and the next few sections will show specific JS-based techniques for making this quite a reasonable approach. But for myself, with TypeScript’s `async/await` now available, I find that it’s a fair bit easier to use TypeScript instead.

*The *JavaScript* version of the canonical data-retrieval-and-reporting automation task*

```
var RawDataTableName = "PopulationTable";
var OutputSheetName = "Top 10 Growing Cities";

Excel.run(function (context) {
    // Here and elsewhere: Create proxy objects to represent
    // the "real" workbook objects that we'll be working with.
    var originalTable = context.workbook.tables
        .getItem(RawDataTableName);

    var nameColumn = originalTable.columns.getItem("City");
    var latestDataColumn = originalTable.columns.getItem(
        "7/1/2014 population estimate");
    var earliestDataColumn = originalTable.columns.getItem(
        "4/1/1990 census population");

    // Now, load the values for each of the three columns that we
    // want to read from.  Note that, to support batching operations
    // together, the load doesn't *actually* happen until
    // we do a "context.sync()", as below.

    nameColumn.load("values");
    latestDataColumn.load("values");
    earliestDataColumn.load("values");

    return context.sync()
        .then(function () {
            // Create an in-memory data representation, using an
            // array with JSON objects representing each city.
            var citiesData = [];

            // Start at i = 1 (that is, 2nd row of the table --
            // remember the 0-indexing) to skip the header.
            for (var i = 1; i < nameColumn.values.length; i++) {
                var name = nameColumn.values[i][0];

                // Note that because "values" is a 2D array
                // (even if, in this case, it's just a single
                //  column), extract the 0th element of each row.
                var pop1990 = earliestDataColumn.values[i][0];
                var popLatest = latestDataColumn.values[i][0];

                // A couple of the cities don't have data for 1990,
                // so skip over those.
                if (isNaN(pop1990) || isNaN(popLatest)) {
                    console.log('Skipping "' + name + '"');
                }

                var growth = popLatest - pop1990;
                citiesData.push({ name: name, growth: growth });
            }

            var sorted = citiesData.sort(function(city1, city2) {
                return city2.growth - city1.growth;
                // Note the opposite order from the usual
                // "first minus second" -- because want to sort in
                // descending order rather than ascending.
            });
            var top10 = sorted.slice(0, 10);

            // Now that we've computed the data, create
            // a report in a new worksheet:
            var outputSheet = context.workbook.worksheets
                .add(OutputSheetName);
            var sheetHeaderTitle = "Population Growth 1990 - 2014";
            var tableCategories = ["Rank", "City", "Population Growth"];

            var reportStartCell = outputSheet.getRange("B2");
            reportStartCell.values = [[sheetHeaderTitle]];
            reportStartCell.format.font.bold = true;
            reportStartCell.format.font.size = 14;
            reportStartCell.getResizedRange
                (0, tableCategories.length - 1).merge();

            var tableHeader = reportStartCell.getOffsetRange(2, 0)
                .getResizedRange(0, tableCategories.length - 1);
            tableHeader.values = [tableCategories];
            var table = outputSheet.tables.add(
                tableHeader, true /*hasHeaders*/);

            for (var i = 0; i < top10.length; i++) {
                var cityData = top10[i];
                table.rows.add(
                    null /* null means "add to end" */,
                    [[i + 1, cityData.name, cityData.growth]]);

                // Note: even though adding just a single row,
                // the API still expects a 2D array (for
                // consistency and with Range.values)
            }

            // Auto-fit the column widths, and set uniform
            // thousands-separator number-formatting on the
            // "Population" column of the table.
            table.getRange().getEntireColumn().format.autofitColumns();
            table.getDataBodyRange().getLastColumn()
                .numberFormat = [["#,##"]];


            // Finally, with the table in place, add a chart:
            var fullTableRange = table.getRange();

            // For the chart, no need to show the "Rank", so
            // only use the column with the city's name -- and
            // then expand it one column to the right
            // to include the population data as well.
            var dataRangeForChart = fullTableRange
                .getColumn(1).getResizedRange(0, 1);

            var chart = outputSheet.charts.add(
                Excel.ChartType.columnClustered,
                dataRangeForChart,
                Excel.ChartSeriesBy.columns);

            chart.title.text =
                "Population Growth between 1990 and 2014";

            // Position the chart to start below the table,
            // occupy the full table width, and be 15 rows tall
            var chartPositionStart = fullTableRange
                .getLastRow().getOffsetRange(2, 0);
            chart.setPosition(chartPositionStart,
                chartPositionStart.getOffsetRange(14, 0));

            outputSheet.activate();
        })
        .then(context.sync);

}).catch(function (error) {
    console.log(error);
    // Log additional debug information, if applicable:
    if (error instanceof OfficeExtension.Error) {
        console.log(error.debugInfo);
    }
});
```

Try it out

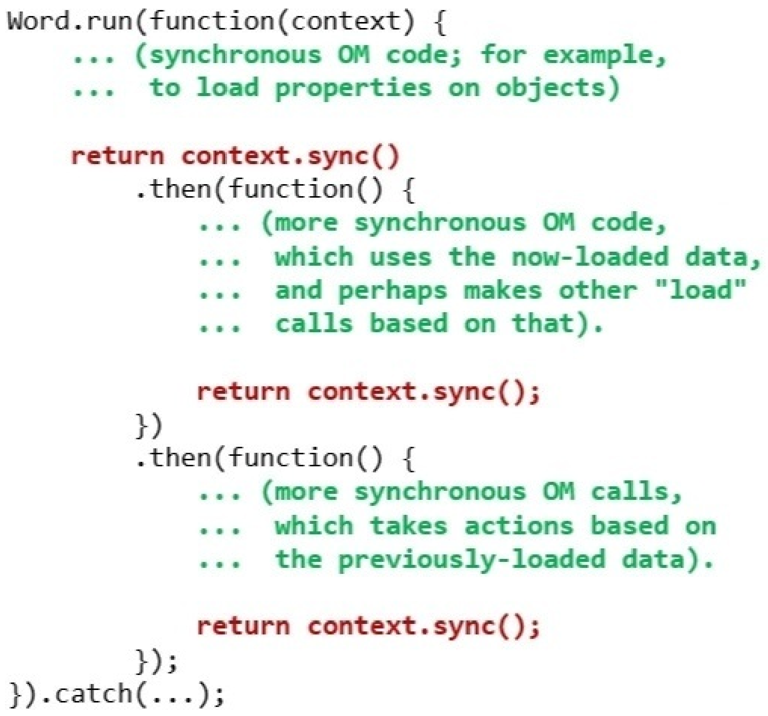
If you want to follow along, just import the following Snippet ID into Script Lab:
98d04bc5293e027c84c8c03741698a94

13.3 JavaScript-specific sync concepts

13.3.1 A JavaScript-based multi-sync template

Section “*De-mystifying* context.sync()” showed an example of what a TypeScript-based pattern for multiple syncs would be.

For JavaScript, the concept is very much the same – and the goal is still to save on the number of sync-s. But instead of await-ing the sync operation, the context.sync() calls must be ***returned*** instead – and the chaining be accomplished using .then functions:



```
Word.run(function(context) {
    ... (synchronous OM code; for example,
    ...  to load properties on objects)

    return context.sync()
        .then(function() {
            ... (more synchronous OM code,
            ...  which uses the now-loaded data,
            ...  and perhaps makes other "load"
            ...  calls based on that).

            return context.sync();
        })
        .then(function() {
            ... (more synchronous OM calls,
            ...  which takes actions based on
            ...  the previously-loaded data).

            return context.sync();
        });
}).catch(...);
```

13.3.2 Returning the context.sync() promise

The previous section showed a JavaScript-based skeleton for a series of Office.js operations. Before we go further, it’s worth pointing out a common pitfall and a way to get around it. Because a batch will typically contain two or more `sync` calls – each of which returns a Promise – it is pivotal *NOT* to break the Promise chain, or else you may encounter some unexpected and often hard-to-diagnose behavior (see "*Chaining Promises, the right way*"). This means that:

1. You must remember to **return** `context.sync()` out of the batch function.
2. You must also remember that the body of each `.then` **must either be fully synchronous, *or* return a Promise**.

#1 is fairly straightforward. In fact, if your omit the initial `return`, as long as you've added a `.catch` statement, you will see the following error during runtime.

> ***RunMustReturnPromise:*** *The batch function passed to the ".run" method didn't return a promise. The function must return a promise, so that any automatically-tracked objects can be released at the completion of the batch operation. Typically, you return a promise by returning the response from* `context.sync()`.

Remembering **#2**, on the other hand, takes more effort. There is unfortunately *no error-checking* that can catch the issue for you, as it's perfectly reasonable (and very common) to have a `.then`-function that doesn't return any values (i.e., is `void`). And moreover, when you do forget the `return` statement, you end up in non-deterministic territory, where the code *might or might not work* based on timing; and where diagnosing the issue is notoriously difficult (with a broken Promise chain, you also lose error-handling; yay!).

Having unintentionally broken the Promise chain myself, on a number of occasions – and having seen beginner Office.js developers break the Promise chain time and time again – I personally prefer to sidestep the issue altogether, using the technique covered in "*Passing in functions to Promise* `.then`*-functions*". Namely, I make *all* functions that make OM calls be *synchronous* (i.e., only queue up the operations, but not dispatch the `context.sync()` yet), and then I follow up these synchronous "`.then`"s with corresponding `.then(context.sync)`-s. Notice how the latter does **not** invoke `context.sync` (there are no `()` after `sync`, as you're passing in a function reference, not a return value). And you can't be guilty of forgetting a return statement, if there is physically no return statement to begin with!

In case the above wasn't 100% clear, let me take a moment to "derive" this transformation:

Let's start at the beginning. Imagine you have code like:

```
...
return context.sync()
    .then(function() {
        // ... synchronous OM code

        return context.sync();
    })
    .then(...)
```

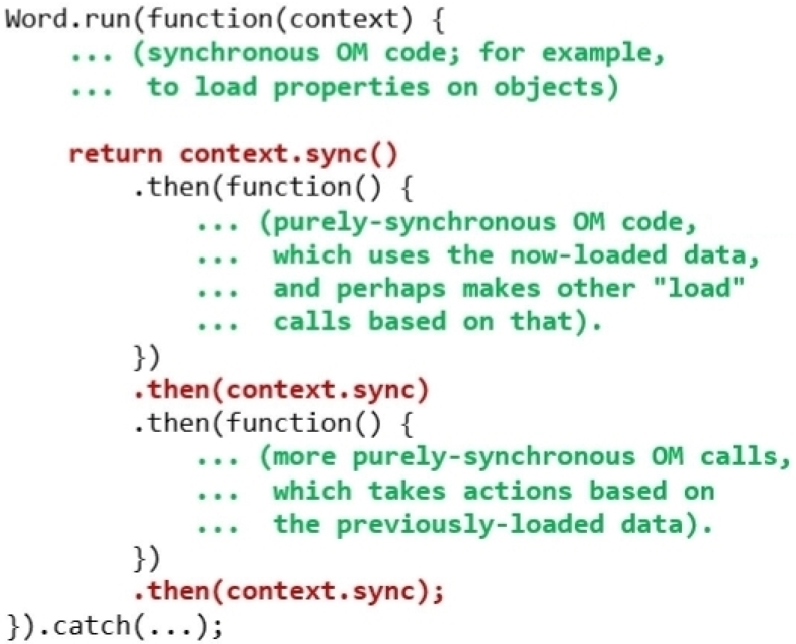
This code makes some OM calls that queue up some operations, and then it invokes context.sync() in an "invoke-style" sync. If we wanted to, we could split out the two parts of the code into two consecutive ".then"s: one for the synchronous OM code, and one for the asynchronous sync invocation. There isn't much *sense* in doing this, but it's just an intermediary process in our derivation:

```
...
return context.sync()
    .then(function() {
        // ... synchronous OM code
    })
    .then(function() {
        return context.sync();
    })
    .then(...)
```

Now, all that the second .then does is create an anonymous function whose sole purpose is to return the result of an invocation of *another* [named] function. This is needlessly complex. Since functions are *first-class citizens* in JavaScript, we can simplify the second .then to take the context.sync function directly, "reference-style", without an invocation!

```
...
return context.sync()
    .then(function() {
        // ... synchronous OM code
    })
    .then(context.sync)
    .then(...)
```

Putting this all together, it means that instead of all the “invoke-style” calls – with the risk of forgetting a return-statement – in the illustration in the preceding section, you can now use the “reference-style” approach to dispatch the sync in a safer manner:



```
Word.run(function(context) {
    ... (synchronous OM code; for example,
    ...  to load properties on objects)

    return context.sync()
        .then(function() {
            ... (purely-synchronous OM code,
            ...  which uses the now-loaded data,
            ...  and perhaps makes other "load"
            ...  calls based on that).
        })
        .then(context.sync)
        .then(function() {
            ... (more purely-synchronous OM calls,
            ...  which takes actions based on
            ...  the previously-loaded data).
        })
        .then(context.sync);
}).catch(...);
```

If you follow the transformation prescribed by this pattern, you’ll find that typically, the only return statement you’ll need is the first return context.sync() invocation – which, between the TypeScript compiler and the runtime error, should be relatively straightforward. For the rest, the problem is avoided altogether via the “reference-style” approach.

13.3.3 Passing values across `.then`-functions

If you’re doing more than two `sync`s as part of your `Excel.run` batch, you’ll often need to pass in a variable from one `.then` to another. I am explaining this here, before showing an end-to-end example, because you can’t do the example *without* having to pass some values across the `.then` boundaries.

First, let’s look at an example where you *don’t* have a problem: the only-two-`sync`-functions case:

*The **successful** case: variables declared outside (before) the first ‘return context.sync()‘ can be accessed within a ‘.then‘*

```
Word.run(function(context) {
    var selection = context.document.getSelection();
    selection.load(...)

    return context.sync()
        .then(function() {
            // The "selection" object is in scope (it was defined
            // at a broader scope that this function shares),
            // so you can use it here all you want:

            if (selection.values === ...) {
                ...
            }
        })
        .then(context.sync);
}).catch(...);
```

The problem begins to manifest itself when you have three or more `sync`-functions, which – if you follow the pattern described in the previous section – get punctuated by two or more “`.then`”s. The problem is that there isn’t a seamless way to pass data from one `.then` to another: the variables declared within a given `.then` are out-of-scope for the next `.then`:

**The problem case:** variables going out of scope

```
Excel.run(function(context) {
    var selection = context.workbook.getSelectedRange();
    selection.load(...)

    return context.sync()
        .then(function() {
            // Based on reading the selection values,
            // choose to load some other data
            var subsetRange = selection.getColumn(...);
            subsetRange.load(...);
        })
        .then(context.sync)
        .then(function() {
            if (subsetRange.values[0][0] = ...) {
                ...
            }

            // The above "subsetRange.values" call will fail,
            // because the earlier assignment of subsetRange
            // is out of scope. You will get an error:
            // "ReferenceError: 'subsetRange' is undefined"
        });
}).catch(...);
```

There are several solutions to this problem.

APPROACH #1: Instead of chaining the “`.then`”s as siblings, nest them instead. The main disadvantage is that the code gets more and more indented, a-la the callback / pyramid-of-doom style. Moreover, you **must not forget the `return` keyword before each `context.sync()` invocation**, or else you end up in non-deterministic territory. BUT, this *does* work, and – if used sparingly – can sometimes be the most elegant solution.

*Approach **#1**: Nesting ‘.then‘s to preserve scope*

```
Excel.run(function(context) {
    var selection = context.workbook.getSelectedRange();
    selection.load(...);

    return context.sync()
        .then(function() {
            var subsetRange = selection.getColumn(...);
            subsetRange.load(...);

            return context.sync()
                .then(function() {
                    if (subsetRange.values[0][0] = ...) {
                        ...
                    }
                });
        });
}).catch(...);
```

APPROACH #2: Declare your variables before the first `return context.sync()`, so that they remain in scope. This is not a bad option. You have to remember to do your variable declarations a fair distance above where you're going to be using them, but that's not too bad. The main disadvantage is IntelliSense: having a `var subsetRange` declaration in one place, and the assignment in quite another, will render the JavaScript IntelliSense experience useless, as the VS IntelliSense engine will no longer be able to deduce the variable's type. On the other hand, **if you're using TypeScript**, you can annotate the variable declaration as `var subsetRange: Excel.Range`, and thereby gain your IntelliSense back. Personally, I find this option much more palatable than the first.

*Approach **#2**: Declaring variables ahead of the first ‘return context.sync()‘ to preserve scope*

```
Excel.run(function(context) {
    var selection = context.workbook.getSelectedRange();
    selection.load(...);

    // Declaring the variable ahead of its use
    var subsetRange;

    return context.sync()
        .then(function() {
            // Assigning to the previously-declared,
            // scope-defying variable:
            subsetRange = selection.getColumn(...);
            subsetRange.load(...);
        })
        .then(context.sync)
        .then(function() {
            // Hooray! "subsetRange" can now be used!
            if (subsetRange.values[0][0] = ...) {
                ...
            }
        });
}).catch(...);
```

APPROACH #3: Pass variables through on the return value. If you have a variable like `subsetRange` inside a `.then`, you can do a `return subsetRange`, close off the function, follow it up with a `.then(context.sync)`, and have the value be passed-through as a parameter to the next function!

*Approach **#3**: Passing variable **through** ‘context.sync‘*

```
 1  Excel.run(function(context) {
 2      var selection = context.workbook.getSelectedRange();
 3      selection.load(...);
 4  
 5      return context.sync()
 6          .then(function() {
 7              var subsetRange = selection.getColumn(...);
 8              subsetRange.load(...);
 9              return subsetRange;
10          })
11          .then(context.sync)
12          .then(function(subsetRange) {
13              // Hooray! "subsetRange" can now be used!
14              if (subsetRange.values[0][0] = ...) {
15                  ...
16              }
17          });
18  }).catch(...);
```

There are several things worth noting with this approach:

- This approach is particularly convenient if all you’re doing is getting a single API object and loading it (as is the case above). In that case, because calling `.load` on the object returns the object back for convenience, you can collapse all three lines (#7-9) down into one: `return selection.getColumn(...).load(...)`.

- The naming of the *passed-in* variable (line #12) need not match the naming in the previous `.then` (though I generally *would* make them match; coming up with a good variable name is hard enough, so why do it twice?!). Obviously, if you are returning the object as per the suggestion above, having never named it, the point becomes moot – and so the incoming parameter name *is* the place where you name it.

- If you want to pass multiple objects, you can. Admittedly, at that point it gets a little messy, so I personally would switch back to **Approach #2** for this. *But*, you can return a complex object, as in

  ```
  return { subsetRange: subsetRange, table: table }
  ```

  On the receiving end of the subsequent `.then`, just declare the function as accepting a `data` parameter, and access the passed-in objects as "`data.subsetRange`", "`data.anotherObject`", etc.

- Unfortunately, as with the previous approach, the VS JavaScript IntelliSense engine is unable to deduce the object's type (TypeScript, on the other hand, *is* able to infer the type – so you don't even need to do a "`var range: Excel.Range`" annotation!)

- If you're using the "invoke-style" `return context.sync()` syntax, instead of the "reference-style `.then(context.sync)` syntax, you can still achieve the same effect by passing the value in as the optional pass-through argument to `context.sync`. That is, you can do

  ```
  return context.sync(subsetRange).
  ```

Finally, **APPROACH #4:** Use the **async/await** feature of TypeScript 2.1… and sidestep the nesting problem altogether.

Personally, TypeScript's async/await feature aside, I prefer to use approach #3 for single-variable pass-through's, and #2 for everything else. However, I should note that I do most of my web coding in TypeScript, where type information can either be inferred or declared. If I were doing pure JS development, I would probably switch to approach #1, just for IntelliSense's sake… or, better yet, convert the project to TypeScript, to reap the IntelliSense benefits and more!

13.3.4 JavaScript example of multi-`sync` calls

The code below shows the *pure-ES5-JavaScript* approach to the scenario described in section “*Real-world example of multiple* `sync`*-calls*”. The automation tasks will filter out all rows and columns, exposing only the row with the student, and only the columns where he/she received less-than-stellar grades.

A *JavaScript-based* three-‘sync‘ automation task

```
Excel.run(function(context) {
    // #1: Find the row that matches the name of the student:

    var sheet = context.workbook.worksheets.getActiveWorksheet();
    var nameColumn = sheet.getRange("A:A");

    var studentName = $("#student-name").val();
    var matchingRowNum = context.workbook.functions.match(
        studentName, nameColumn, 0 /*exact match*/);
    matchingRowNum.load("value");

    var studentRow; // declared here for passing between "`.then`"s

    return context.sync()
        .then(function() {
            // #2: Load the cell values (filtered to just the
            //     used range, to minimize data-transfer)

            studentRow = sheet.getCell(matchingRowNum.value - 1, 0)
                .getEntireRow().getUsedRange();
            studentRow.load("values");
        })
        .then(context.sync)
        .then(function() {
            // Hide all rows except header ones and the student row
            var cellB2AndOnward = sheet.getUsedRange()
                .getOffsetRange(1, 1);
            cellB2AndOnward.rowHidden = true
            cellB2AndOnward.columnHidden = true;
            studentRow.rowHidden = false;
31
32              for (var c = 0; c < studentRow.values[0].length; c++) {
33                  if (studentRow.values[0][c] < 80) {
34                      studentRow.getColumn(c).columnHidden = false;
35                  }
36              }
37
38              studentRow.getCell(0, 0).select();
39          })
40          .then(context.sync);
41  }).catch(OfficeHelpers.Utilities.log);
```

Note the **`return context.sync()`** call on line **#14** (where the `return` statement ensures that the value returned out of the batch function is a Promise), and then the two **`.then(context.sync)`**-s on lines **#23** and **#40**.

Download sample code

To view or download this code, follow this link:
http://www.buildingofficeaddins.com/samples/gradebook

While the code above accomplishes the original scenario, it's worth pointing out (am I beating a dead horse?) some aspects that are less ideal compared to the TypeScript version:

1. **Nested "`.then`"s**. The back-and-forth indentation made the code more tedious to write and harder to read. It also forced the unnatural separation of variable declarations from usage (I declared all my variables at the very top – see lines **#6-12** – to allow me to use them across the "`.then`"s).

2. A big side-effect of #1: **Lack of reliable IntelliSense**. By declaring the variables at the very top of `Excel.run` – separately from assigning them – I lost IntelliSense for most of my objects. There are workarounds for this (see "*JavaScript IntelliSense*"), but they require extra work.

3. Lack of compile-time safety. Especially when dealing with a complex OM, and especially when IntelliSense is spotty, it's very easy to introduce errors into the code: anything from misspelling a variable, to passing invalid parameters into a method, to assigning an incorrect object type to a property.

TypeScript solves all three of these problems magnificently:

- For **reliable IntelliSense and compile-time safety**, TypeScript offers not only a better type-inferencing engine, but it also offers a way to declaratively assert a variable's type, as in

  ```
  let stockNamesRange: Excel.Range
  ```

  This both aids the IntelliSense engine, and allows the compiler to catch misspellings, type errors, and so forth.

- For **avoiding nested "`.then`"-functions, `async/await` does away with the problem altogether**. Coincidentally, the de-nesting of the code also helps the type-inferencing engine, so that even appending a type annotation becomes generally unnecessary.

14. Appendix B: Miscellanea

Chapter structure

14.1 Script Lab: the story behind the project

I love playgrounds! When I first was learning JavaScript, I used JSFiddle – the *classic* playground of the web – a dozen times each day. But for me, a Playground isn't just a point along the learning journey; I continued to return to JSFiddle day in an day out, even as a well-versed web developer, because *it was so freakin' easy* to try out something new, on the spot, without having to mess up my existing project. And if I had a question, or an answer to a StackOverflow question, it allowed for easy sharing: "see this fiddle for a complete working sample".

For a company-wide hackathon project in the summer of 2016, myself and two other colleagues – Bhargav Krishna and David Nissimoff – decided to build a playground for Office Add-ins. The idea was actually inspired by my book: the more code samples I wrote, the more I wished for an easy tool where readers could try out small nuggets of code with minimal effort. But while the book has been a completely moon-lighted, non-MSFT-affiliated project, the playground was something we were doing as part of work. Though the effort was completely grass-root, sparked by the 2.5-day hackathon and then continued as a labor of love on our spare (and sometimes work) time, our management bought off on our vision and gave Bhargav and I – and then, a couple of newly-joined members as well – the time necessary to see this project through. And that is how Script Lab came to be.

Script Lab offers a bunch of functionality. But most importantly, it will allow you to:

- Explore sample snippets, hand-picked by the team to show the best coding practices
- Create your own snippets, keeping them as part of your workspace (and being able to switch back-and-forth between snippets, copy-pasting in sample content, etc.,)
- Import existing snippets (from StackOverflow, etc.), and also share snippets as part of StackOverflow questions or answers.

In addition to its functional use as a prototyping/playground tool, Script Lab also shows the best of what we consider to be modern coding practices and technologies. We use Angular 4, TypeScript, the Node & NPM ecosystem (with dozens of libraries), the Office Fabric JS library, Handlebars (for templating), WebPack (for munging together and bundling the source code), Browser-Sync (for instant browser refreshes whenever anything changes), and every other technology that Bhargav could throw at it. Script Lab also exemplifies some of the design patterns that we encourage add-in developers to use (e.g., the loading splash screen, the hamburger-based navigation, the iconography, etc.) With the project open-source, we encourage folks to take a look at how we built things – and if they have suggestions or features they want to contribute, all the better!

Script Lab is brought to you by this group of good-looking geeks (with myself on bottom right). Photo from https://aka.ms/scriptlab.

To read more about Script Lab, or to visit links to videos and other media content about the project, see the project's GitHub repository and README: https://aka.ms/script-lab